CP/M-8000™

Operating System

Programmer's Guide

COPYRIGHT

Copyright © 1984 Digital Research Inc. All rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without the prior written permission of Digital Research Inc., 60 Garden Court, Box DRI, Monterey, California 93942.

DISCLAIMER

DIGITAL RESEARCH INC. MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. Further, Digital Research Inc. reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Digital Research Inc. to notify any person of such revision or changes.

NOTICE TO USER

From time to time changes are made in the filenames and in the files actually included on the distribution disk. This manual should not be construed as a representation or warranty that such files or facilities exist on the distribution disk or as part of the materials and programs distributed. Most distribution disks include a "README.DOC" file. This file explains variations from the manual which do constitute modification of the manual and the items included therewith. Be sure to read this file before using the software.

TRADEMARKS

CP/M, CP/M-86, CP/NET, and Digital Research and its logo are registered trademarks of Digital Research, Inc. Concurrent, CP/M-68K, CP/M-8000, DDT, MP/M, and MP/M-86 are trademarks of Digital Research, Inc. IBM is a registered trademark of International Business Machines. Olivetti is a registered trademark of the Olivetti Corporation. Zilog is a registered trademark of Zilog, Inc.. AR8K, ASZ8K, LD8K, ZCC, and Z8000 are trademarks of Zilog, Inc.

The CP/M-8000 Operating System Programmer's Guide was printed in the United States of America.

```
************************************
*  First Edition:  October 1984  *
************************************
```

Foreword

CP/M-8000™ is a single-user operating system designed for the Zilog® Z8000™ microprocessors. CP/M-8000 requires a minimum of 128K bytes of random access memory (RAM) to run its base-level system, which contains the following CP/M® commands and utilities:

- CP/M Built-in Commands:

  DIR
  DIRS
  ERA
  REN
  SUBMIT
  TYPE
  USER

- Standard CP/M Utilities:

  ED
  PIP
  STAT

- Programming Utilities:

  Archive (AR8K™)
  DUMP
  SIZEZ8K
  XDUMP

- Programming Tools

  Assembler (ASZ8K™)
  DDT™
  Linker/Loader (LD8K™)
  Symbol Table Print Utility (NMZ8K)
  C Compiler (ZCC™, Z8K™, ZCC1, ZCC2, and ZCC3)*

* Described in the C Language Programmer's Guide for CP/M-8000.

The CP/M-8000 file system is based on and upwardly compatible with the CP/M Release 2.2, CP/M-86® Release 1.1, and CP/M-68K™ Release 1.2 file systems. However, CP/M-8000 supports a much larger file size with a maximum of 32 megabytes per file.

CP/M-8000 supports a maximum of 16 disk drives, with 512 megabytes per drive. CP/M-8000 supports other peripheral devices that the Basic I/O System (BIOS) assigns to one of the four logical devices: LIST, CONSOLE, AUXILIARY INPUT, or AUXILIARY OUTPUT.

Organization Of This Guide

This guide is organized into the following sections:

Section 1 contains an overview of the operating system's architecture, including descriptions of file system access, programming tools, file specifications, and the terminology used in this guide.

Section 2 describes the Console Command Processor (CCP), the mechanisms CP/M-8000 uses for loading and exiting transient programs, and a program execution model.

Section 3 presents the CP/M-8000 command file format.

Section 4 documents the Basic Disk Operating System (BDOS) functions.

Section 5 describes the operation and commands of the AS28K Assembler.

Section 6 describes the operation and commands of the CP/M-8000 Linker, LD8K.

Section 7 describes the command lines that invoke and output from the AR8K, DUMP, XDUMP, SIZEZ8K, and XCON utilities.

Section 8 describes the operation and commands of DDT.

Before You Use This Manual

The presentation of information in this guide assumes you are an experienced programmer familiar with the basic programming concepts of assembly language. If you are not familiar with Zilog Z8000 assembly language, refer to the following manuals:

- Z8000 CPU User's Reference Manual, Prentice-Hall, 1982
- Z8000 CPU Programmer's Guide, Zilog (611-1790-0006), 1981
- Mateosial, Richard. Programming the Z8000, Sybex, 1980

Before you can use the facilities in this guide, your CP/M-8000 system must be configured for your particular hardware environment. Normally, your system is configured for you by the manufacturer of your computer or the software distributor. If you have an unusual hardware environment, however, this may not be the case. Refer to the CP/M-8000 Operating System System Guide (cited as CP/M-8000 System Guide for details on how to oonfigure your system for a custom hardware environment.

Command Conventions Used In This Guide

The following list describes the command conventions used in this guide.

[] — Square brackets in a command line enclose optional parameters.

nH — The capital letter H follows numeric values that are represented in hexadecimal notation.

numeric values — Numeric values are represented in decimal notation unless stated otherwise.

(n) — BDOS function numbers are enclosed in parentheses when they appear in text.

⋮ or . . . — Unless noted otherwise, a vertical or horizontal ellipsis indicates missing elements in a series.

RETURN — The word RETURN refers to the Return or Enter key on the keyboard of your console. Unless otherwise noted, you must press Return to invoke a command line entered from your console.

CTRL-X — CTRL-X indicates that you must hold down the Control key while simultaneously pressing the key indicated by the variable X.

New Functions and Implementation Changes

CP/M-8000 has six new Basic Disk Operating System (BDOS) functions and additional implementation changes in the BDOS functions and data structures that differ from other CP/M systems. The new BDOS functions and implementation changes are listed in Appendix F.

Table F-4 in Appendix F contains the functions and commands supported by other CP/M systems that are not supported by CP/M-8000.

Table of Contents

Table of Contents (continued)

4 **Basic Disk Operating System Functions**

Table of Contents (continued)

Table of Contents
(continued)

Table of Contents (continued)

Appendixes

Table of Contents
(continued)

Tables, Figures, and Listings

Tables

Tables, Figures, and Listings (continued)

Tables, Figures, and Listings (continued)

Tables, Figures, and Listings (continued)

Figures

Listings

Section 1
Introduction to CP/M-8000

CP/M-8000 has many of the facilities of other CP/M systems. It also contains features which allow it to address up to sixteen megabytes of main memory available on the Z8000 microprocessor. The CP/M-8000 file system is upwardly compatible with CP/M Release 2.2, CP/M-86 Release 1.1, and CP/M-68K Release 1.2. The CP/M-8000 file structure supports up to sixteen 512 megabyte disk drives and a maximum file size of 32 megabytes.

1.1 CP/M-8000 Architecture

The CP/M-8000 operating system is contained in the CPM.SYS file on the system disk. CPM.SYS is loaded into memory during a cold start by a cold start loader module that resides on the first three tracks of the system disk. Table 1-1 lists the three program modules contained in CPM.SYS.

Table 1-1. Program Modules in the CPM.SYS File

Module	Mnemonic	Description
Console Command Processor	CCP	parses user command lines
Basic Disk Operating System	BDOS	provides file system access functions
Basic I/O System	BIOS	provides the device-driving routines for peripheral I/O

The sizes of the CCP and BDOS modules are fixed for a given release of CP/M-8000. The size of the BIOS custom module, normally supplied by the computer manufacturer or software distributor, depends upon the system configuration, which varies with the implementation.

The specific implementation of CP/M-8000 dictates which memory segment is loaded with the operating system. In some implementations, for example, the system kernel may use nonsegmented system address space (to avoid the extra memory required for code/data separation). The configuration of physical memory within specific Z8000-based computer systems also affects which memory segment the operating system can be loaded into for execution. All CP/M-8000 modules remain resident in memory. The CCP cannot be used as a data area subsequent to transient program load.

1.2 Transient Programs

The Transient Program Area (TPA) is made up of the remaining segments of address space that are not occupied by the operating system after CP/M-8000 is loaded into memory. CP/M-8000 loads executable files, called command files, from disk into the TPA. Because they are temporarily, rather than permanently, resident in memory (and hence not configured within CP/M-8000), these command files are also called transient commands or transient programs. The CP/M-8000 command file format is described in Section 3.

Nonsegmented transient programs can run in a single TPA segment or in two segments: one for instructions and one for data (called split I and D spaces). Segmented programs can use any segment of the TPA specified in their object files.

1.3 File System Access

Programs do not specify absolute locations or default variables when accessing CP/M-8000. Instead, programs invoke BDOS and BIOS functions. Section 4 describes the BDOS functions in detail. Appendix A lists the BIOS calls. Refer to the CP/M-8000 Operating System System Guide for detailed descriptions of the BIOS functions.

In addition to these functions, CP/M-8000 decreases dependence on absolute addresses by maintaining a base page in the TPA for each transient program in memory. The base page contains initial values for the File Control Block (FCB) and the Direct Memory Access (DMA) buffer. (See Section 2.2 for details on the base page and methods for loading transient programs.)

1.4 Programming Tools and Commands

CP/M-8000 programming tools and utilities include an Assembler (ASZ8K), Linker/Loader with relocation capability (LD8K), Archive utility (AR8K), Hex/ASCII DUMP utility, and Object File Dump utility (XDUMP). Table 1-2 lists the commands that invoke each of these tools and the section of this guide in which the programming tool is described. Tables 1-2, 1-3, and 1-4 list other commands supported by CP/M-8000 and the manuals in which they are documented.

Table 1-2. CP/M-8000 Programmer's Guide Commands

Command	Description
AR8K	Invokes the Archive Utility (AR8K). AR8K creates a library and/or deletes, adds, or extracts object modules from an existing library, such as the C Run-time Library. AR8K is described in Section 7.
ASZ8K	Invokes the CP/M-8000 Assembler (ASZ8K). ASZ8K is described in Section 5.
DDT	Invokes the CP/M-8000 version of DDT, the Dynamic Debugging Tool. Section 8 describes DDT.
DUMP	Invokes the DUMP utility that prints the contents of a file in hexadecimal and ASCII notation. DUMP is described in Section 7.2.
LD8K	Invokes the Linker/loader that combines several assembled (object) programs into one executable command file and creates absolute files from relocatable command files. This utility is described in Section 6.
NMZ8K	Invokes the NMZ8K utility that prints the symbol table of an object or command file. The use of NMZ8K is described in Section 3.5, "Printing the Symbol Table."
SIZEZ8K	Invokes the SIZEZ8K utility that displays the total size of a command file and the size of each of its program segments. SIZEZ8K is described in Section 7.4.
XCON	Invokes the XCON utility that converts ASZ8K object output into the x.out format. The x.out format is described in Section 3; Section 7.5 describes XCON.
XDUMP	Invokes the XDUMP utility that prints the header, contents, and symbol table of an object or command file. Section 7.3 describes this utility.

Table 1-3 briefly describes the commands documented in the C/PM-8000 Operating System User's Guide (cited as CP/M-8000 User's Guide).

Table 1-3. CP/M-8000 User's Guide Commands

Command	Description
COPY	copies disks (including the boot tracks)
DIR*	displays the directory of files on a specified disk
DIRS*	displays the directory of system files on a specified disk
ED	invokes the CP/M-8000 text editor
ERA*	erases one or more specified files
FORMAT	prepares floppy disks for use with CP/M-8000
PIP	copies, combines, and transfers specified files between peripheral devices
REN*	renames an existing file to the new name specified in the command line
STAT	shows disk and file access status, free space on disks, file size, or the logical-to-physical device assignments, according to command line options
SUBMIT*	executes a file of CP/M commands
TYPE*	displays the contents of an ASCII file on the console
USER*	displays or changes the current user number

* CP/M-8000 built-in commands

Table 1-4 briefly describes the commands documented in the C Language Programmer's Guide for CP/M-8000 (cited as C Language Programmer's Guide).

Table 1-4. C Language Programmer's Guide Commands

Command	Description
ZCC	invokes a command line interpreter that loads the compiler for the compilation of CP/M-8000 C source files
ZCC1	invokes the preprocessor for processing macros when you compile C source files
ZCC2	invokes the C parser when you compile CP/M-8000 C source files
ZCC3	invokes the code generator optimizer for the CP/M-8000 C compiler when you compile C source files

1.5 CP/M-8000 File Specification

The CP/M-8000 file specification is compatible with other CP/M systems. The format contains three fields: a one-character drive select code (d), a one- to eight-character filename (f...f), and a one- to three-character filetype (ttt) field, as shown in the following example:

```
Format          d:ffffffff.ttt

Example         B:GINA.DAT
```

The drive select code and filetype fields are optional. A colon (:) delimits the drive select field. A period (.) delimits the filetype field. These delimiters are required only when the fields they delimit are specified.

Values for the drive select code range from A through P when the BIOS implementation supports 16 drives, the maximum number allowed. The range for the drive code depends on the BIOS implementation. Drives are labeled A through P to correspond to the 1 through 16 drives supported by CP/M-8000. However, not all BIOS implementations support the full range.

The characters in the filename and filetype fields cannot contain delimiters (the colon and period) and must be uppercase for the CCP to parse the file specification. The CCP cannot access a file that contains delimiters or lowercase characters. When entered at the CCP level, a command line and its file specifications, if any, are internally translated to uppercase before the CCP parses them.

Not all commands and file specifications are entered at the CCP level. CP/M-8000 allows you to include delimiters or lowercase characters in file specifications that are created or referenced by functions that bypass the CCP. For example, the BDOS Make File Function (22) allows you to create a file specification that includes delimiters and lowercase characters, although the CCP cannot parse and access such a file.

Table 1-5 lists some additional delimiter characters you should avoid using in your file specifications. These characters are reserved because several CP/M-8000 built-in commands and utilities have special uses for them.

Table 1-5. Delimiter Characters

Character	Description
[]	square brackets
()	parentheses
<>	angle brackets
=	equals sign
*	asterisk
&	ampersand
,	comma
!	exclamation point
|	bar
?	question mark
/	slash
$	dollar sign
.	period
:	colon
;	semicolon
+	plus sign
-	minus sign

1.6 Wildcards

CP/M-8000 supports two wildcards, the question mark (?) and the asterisk (*). Several utilities and BDOS functions allow you to specify wildcards in a file specification to perform the operation or function on one or more files. However, BDOS functions support only the ? wildcard.

The ? wildcard matches any one character in the character position occupied by this wildcard. For example, the file specification G?NA.DAT indicates the second letter of the filename can be any alphanumeric character if the remainder of file specification matches. Thus, the ? wildcard matches exactly one character position.

The * wildcard matches one or more characters in the field or remainder of a field that this wildcard occupies. CP/M-8000 internally pads the field or remaining portion of the field occupied by the * wildcard with ? wildcards before searching for a match. For example, CP/M-8000 converts the file specification B*.DAT to B???????.DAT before searching for a matching file specification. Thus, any file that starts with the letter B and has a filetype of DAT matches this file specification.

For details on wildcard support by a specific BDOS function, refer to the description of the function in Section 4 of this guide. For additional details on these wildcards and support by CP/M-8000 utilities, refer to the CP/M-8000 User's Guide.

1.7 CP/M-8000 Terminology

Table 1-6 lists the terminology used throughout this guide to describe CP/M-8000 values and program components.

Table 1-6. CP/M-8000 Terminology

Term	Meaning
Nibble	4-bit value
Byte	8-bit value
Word	16-bit value
Longword	32-bit value
Address	32-bit value that specifies a location in storage
Offset	fixed displacement that references a location in storage, other data source, or destination
Text Segment	program section that contains instructions

Table 1-6. (continued)

Term	Meaning
Data Segment	program section that contains initialized data
Block Storage Segment (bss)	program section that contains uninitialized data
Segment (Z8001)	set of adjacent memory addresses (up to 64K) with the same segment number
Segmented Mode	running-state of the segmented CPU in which addresses can have different segment members
Nonsegmented Mode	running-state of the Z8000 CPU -- addresses generated by segmented CPUs in this mode have the same segment number

End of Section 1

Section 2
The CCP and Transient Programs

This section discusses the Console Command Processor (CCP), built-in and transient commands, transient program loading and exiting, and CP/M-8000 memory models.

2.1 CCP Built-in and Transient Commands

After an initial cold start, CP/M-8000 displays a sign-on message at the console. Drive A, containing the system disk, is logged in automatically. The standard prompt (>), preceded by the letter A designating the drive, is displayed on the console screen. This prompt informs the user that CP/M-8000 is ready to receive a command line from the console.

In response to the prompt, a user types the filename of a command file and a command tail, if required. CP/M-8000 supports two types of command files, built-in commands and transient commands. Built-in commands are configured and reside in memory with CP/M-8000. Transient commands are loaded in the TPA and do not reside in memory allocated to CP/M-8000.

CP/M-8000 supports these seven built-in commands:

- DIR
- DIRS
- ERA
- REN
- TYPE
- USER
- SUBMIT

A transient command is a machine-readable, executable program file loaded from disk to memory. Section 3 describes the format of transient command files.

When the user enters a command line, the CCP parses it and tries to load the specified file. The CCP assumes a file is a command file when any filetype other than SUB is specified. When the user specifies only the filename but not the filetype, the CCP searches for and tries to load a file with a matching filename and a filetype of either Z8K or three blanks. The CCP searches the current user number and then user number 0 for a matching file. If this search does not yield a command file, but the CCP finds a matching file with a filetype of SUB, the CCP executes it as a submit file.

2.2 Loading a Program

Either the CCP or a transient program can load a program in memory with the BDOS Program Load Function (59) described in Section 4.5.7. After the program is loaded, the TPA contains the program segments (text, data, and bss), a user stack, and a base page. A base page exists for each loaded program.

The base page is a 256-byte data structure that defines a program's operating environment. The base page in CP/M-8000 does not reside at a fixed absolute address prior to being loaded. The BDOS Program Load Function (59) determines the absolute address of the base page when the program is loaded. The Program Load Function and the CCP or the transient program initialize the contents of the base page and the program's stack as described below.

2.2.1 Base Page Initialization By The CCP

The CCP parses up to two filenames following the command in the command line. The CCP places the properly formatted FCB's in the base page. The default DMA address is initialized at an offset of 0080H in the base page. The default DMA buffer occupies the second half of the base page. The CCP initializes the default DMA buffer to contain the command tail. The format of the command tail is described in Section 2.2.3. The CCP invokes the BDOS Program Load Function (59) to load the transient program before it parses the command line.

Program Load, Function 59, allocates space for the base page and initializes base page values at offsets 0000H through 0024H from the beginning of the base page (see Appendix C). Values at offsets 0025H through 0037H are not initialized, but the space is reserved. The CCP parses the command line and initializes values at offsets 0038H through 00FFH. Before the CCP gives control to the loaded program, it pushes the address of the transient program's base page and a return address, within the CCP, on the user stack. When the program is invoked, the top of the stack contains a return address within the CCP, which is pointed to by the stack pointer, register R15 for nonsegmented programs or RR14 for segmented. The address of the program's base page is located at a 4-byte offset from the stack pointer.

2.2.2 Loading Multiple Programs

Multiple programs can reside in memory, but the CCP can load only one program at a time. However, a transient program, loaded by the CCP, can load one or more additional programs in memory. A program loads another program in memory by invoking the BDOS Program Load Function (59). The CCP supplies FCB's and the command tail to this function. When the CCP is not present the transient program must provide this information, if required, for any additional programs it loads.

2.2.3 Base Page Initialization by a Transient Program

A transient program invokes the BDOS Program Load Function (59) to load an additional program. The BDOS Program Load Function allocates space and initializes base page values at offsets 0000H through 0024H for the program as described in Section 2.2.1. The transient program must initialize the base page values that the CCP normally supplies, such as FCB's, the DMA address, and the command tail, if the program being loaded requires these values. The command tail contains the command parameters but not the command. The format of the command tail in the base page consists of a one-byte character count, followed by the characters in the command tail, and terminated by a null byte as shown in Figure 2-1. The command tail cannot contain more than 126 bytes plus the character count and the terminating null character.

Count	Characters in the Command Tail	0
1 byte	N bytes ≤ 126 bytes	

Figure 2-1. Format of the Command Tail in the DMA Buffer

Unlike the CCP, a transient program does not necessarily push the address of its base page and a return address on the user stack before giving control to the program that it loads with the Program Load Function. The transient program can be designed to push these addresses on the user stack of the program it loads (if the loaded program uses the base page).

The address of the base page for the loaded program is not pushed on the user stack by the Program Load Function (59). Instead, it is returned in the Load Parameter Block (LPB), which is used by the BDOS Program Load Function. Appendix B contains an example of a C language program, PGMLD.C. PGMLD.C illustrates how a transient program loads another program without the CCP, using the BDOS Program Load Function (59). Appendix C summarizes the offsets and contents of a CP/M-8000 base page.

2.3 Exiting Transient Programs

CP/M-8000 supports two ways of exiting a transient program and returning control to the CCP:

- Interactively, by typing CTRL-C at the console, the default I/O device
- A programmed return to the CCP using either of the following:

  1 a Return From Subroutine (RET) Instruction
  2 the BDOS System Reset Function (0)

The CCP will regain control when a user types CTRL-C from the console only if the program uses one of these BDOS functions:

- Console Output (2)
- Print String (9)
- Read Console Buffer (10)

On input, CTRL-C must be the first character that the user types on the line. CTRL-C terminates execution of the main program and any additional programs loaded beyond the CCP level. For example, typing CTRL-C while debugging a program terminates execution of the program being debugged and DDT before the CCP regains control. Typing CTRL-C in response to the system prompt resets the status of all disks to read/write.

To program a return to the CCP, specify the BDOS System Reset Function (0) or a Return From Subroutine (RET) Instruction. In programs written in C, a subroutine return from the main program to the run-time package will cause this function to be executed. Invoking the BDOS System Reset Function (0) is equivalent to programming a return to the CCP. This function performs a warm boot, which terminates the execution of a program before it returns program control to the CCP. The BDOS System Reset Function is described in Section 4.5.1.

The RET instruction must be the last one executed in the program and the top of the stack must contain the system-supplied return address for control to return to the CCP. When a transient program begins execution, the top of the user stack contains this system-supplied return address within the CCP. If the program modifies the stack, the top of the stack must contain this system-supplied return address before the RET instruction is executed.

2.4 Transient Program Execution Model

CP/M-8000 divides memory into two categories: System and the Transient Program Area (TPA). CP/M-8000 System memory contains the Basic Disk Operating System (BDOS), the Basic I/O System (BIOS), and the Console Command Processor (CCP). The bootstrap program initializes the memory locations for these components which can reside in any single segment, provided, the BDOS and CCP are contiguous. CP/M-8000 Exception Vector handling is performed by a subroutine present in the BDOS.

The System memory components (BDOS, BIOS, and CCP) use the Z8000's System Address Space, in which code and data are combined to save the space required for code/data separation. The TPA, which consists of the memory segments not occupied by the operating system, resides in the Z8000's Normal Address Space.

A user stack, a base page, and the three program segments: a text segment, an initialized data segment, and a block storage segment (bss), exist for each transient program loaded in the TPA. The BDOS Program Load Function (59) loads a transient program in the TPA. If memory locations are not specified when the transient program is linked, the program is loaded in the TPA as shown in Figure 2-2.

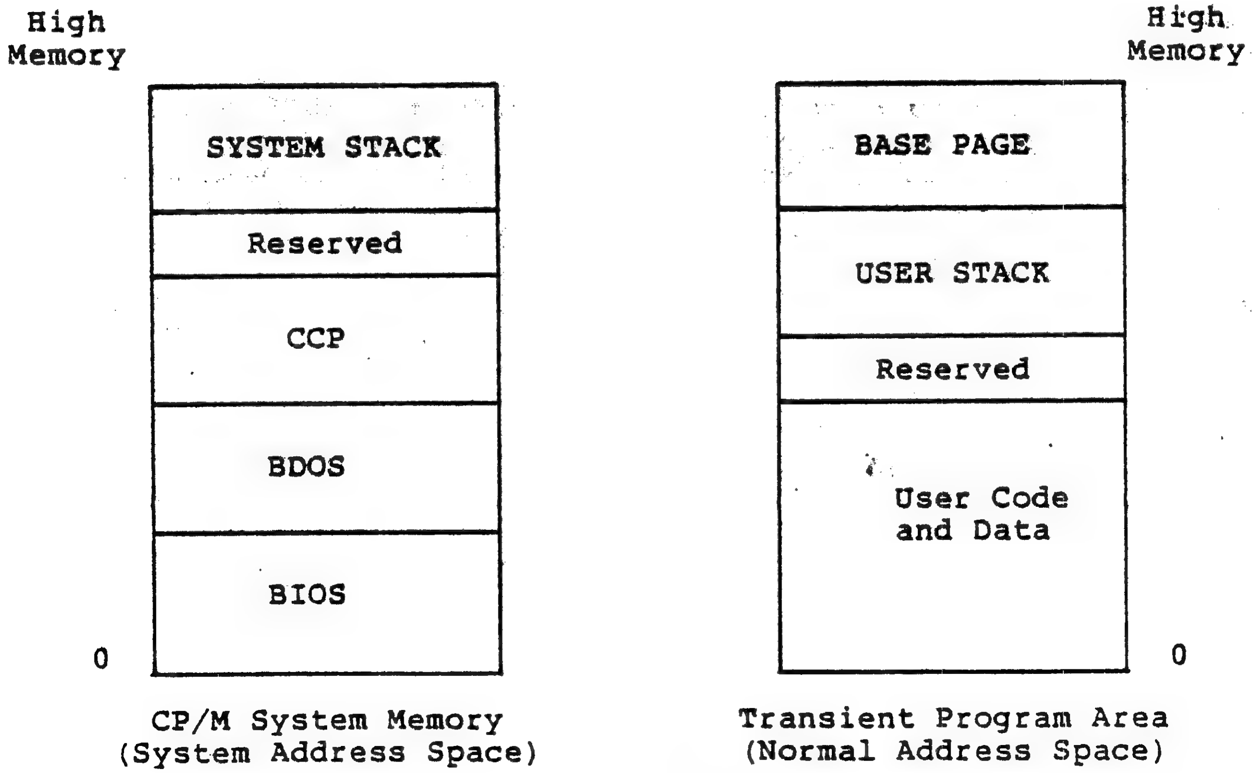


Figure 2-2. CP/M-8000 Default Memory Model

The TPA can be a segmented or nonsegmented area of memory. If the TPA is nonsegmented, it can combine or separate code and data in different segments, depending upon the hardware configuration and the transient program's space requirements.

CP/M-8000 contains three additional BIOS system calls to support memory management and address space communication:

- Map Address (_map_adr)
- Memory Copy (_mem_cpy)
- Transfer Control (_xfer)

The _map_adr system call translates logical addresses into physical addresses. The _mem_cpy system call copies a specified number of bytes from one physical address to another. The _xfer system call transfers control to a new program context. These functions are available to transient programs. For example, a transient program can obtain a copy of the Memory Region Table as follows:

1. Fetch the MRT System Address using the BIOS Function 18.

2. Translate the System Address into a Physical Address with _map_adr.

3. Copy the MRT from the Physical Address into the program's own address space using the _mem_cpy System Call.

Section 4.2 of the CP/M-8000 System Guide describes the BIOS System Calls in detail.

Some systems can configure and load CP/M-8000 in such a manner that one or more portions of high memory cannot be addressed by the CP/M-8000 operating system. In such instances, CP/M-8000 does not know the memory exists and cannot define or configure the memory in the BIOS. However, a transient program that knows this memory exists can access it.

End of Section 2

Section 3
Command File Format

This section describes the x.out format of CP/M-8000 command files. Command files are output by the Linker/Loader (LD8K), Assembler (ASZ8K) through XCON, and C Compiler (ZCC). These utilities give a command file the default file name of X.OUT.

A command file contains a header, segment information array, text and data segments, relocation data, and a symbol table. These components are shown in Figure 3-1 and described in the following sections.

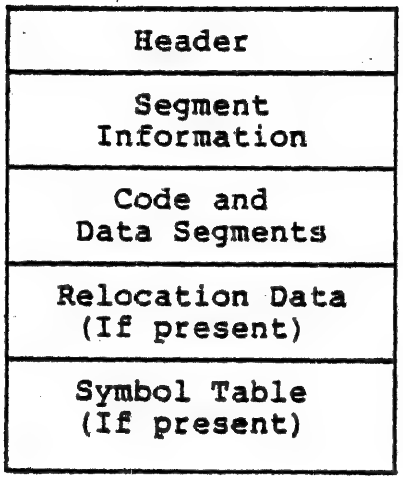
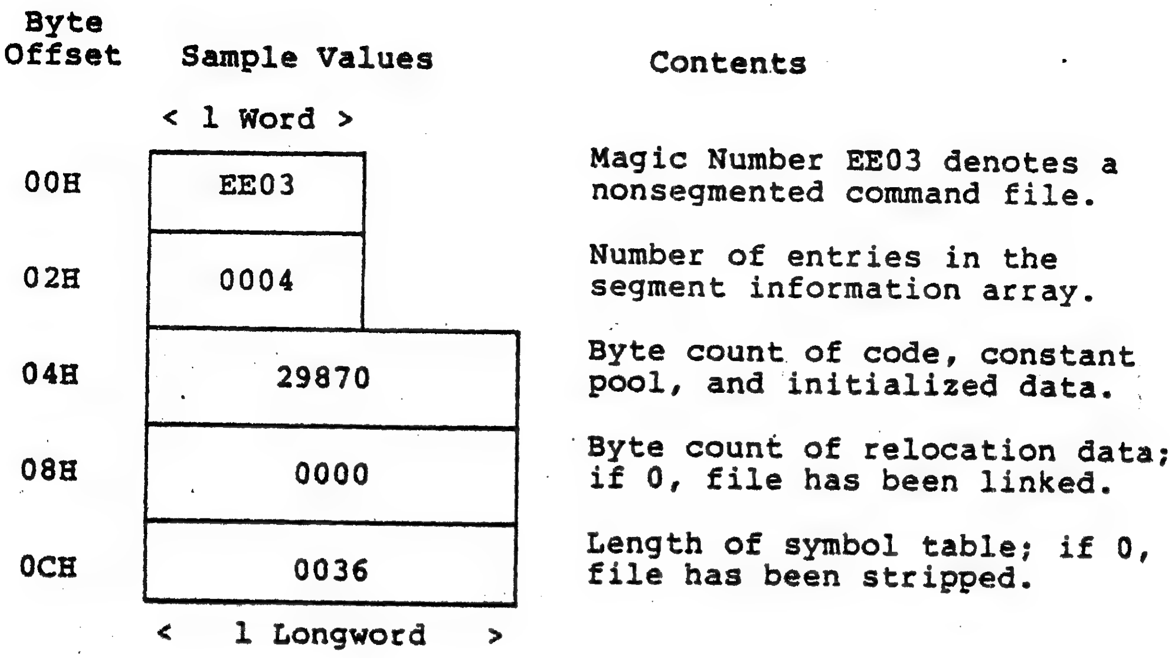


Figure 3-1. CP/M-8000 Command File Layout

3.1 The Command File Header

The first component of a CP/M-8000 command file is the header. It specifies the file's data type and the size and starting address of the file's other components. Figure 3-2 shows the format of a CP/M-8000 command file header.



Byte Offset	Sample Values	Contents
	< 1 Word >	
00H	EE03	Magic Number EE03 denotes a nonsegmented command file.
02H	0004	Number of entries in the segment information array.
04H	29870	Byte count of code, constant pool, and initialized data.
08H	0000	Byte count of relocation data; if 0, file has been linked.
0CH	0036	Length of symbol table; if 0, file has been stripped.
	< 1 Longword >	

Figure 3-2. CP/M-8000 Command File Header Fields

The Magic Number field of the header specifies one of six data types for each CP/M-8000 command file. Valid Magic Numbers are shown in Table 3-1.

Table 3-1. Magic Number Values

Magic Number	Type of Data
EE00	segmented, nonexecutable
EE01	segmented, executable
EE02	nonsegmented, nonexecutable
EE03	nonsegmented, executable, nonshared I & D space
EE07	nonsegmented, executable, shared I & D space
EE0B	nonsegmented, executable, split I & D space

Because of word alignment, the byte counts in the third, fourth, and fifth fields of the header will always be even. The code and data segments, relocation data, and symbol table are word aligned.

Section 3.5 describes how to print the header information of an object or command file with the XDUMP utility.

3.2 Segment Information

The segment information portion of the command file contains an entry for each of the file's segments. Each entry consists of the following four fields:

- The segment's assigned number (one byte): a preassigned number from 0 to 127 or a value of 255. A value of 255 is used to indicate that the linker can assign the segment's number.
- The segment's type (one byte): the type and content of the segment are indicated according to the values shown in Table 3-2.
- The segment's execution length (one unsigned word)
- The segment's array element (one byte): a subcount of the number of segment entries in the second field of the header.

Table 3-2. Segment type values.

Value	Type
1	Uninitialized data segment (bss) -- the corresponding portion of the file is not present
2	Stack segment -- no data in file
3	Code or text segment
4	Constant pool
5	Initialized data
6	Mixed code/data, not protectable
7	Mixed code/data, protectable

Type 1 specifies space for uninitialized data generated by the program during execution. Although space for the bss is specified in the source command file, it is not allocated until the command file is loaded into memory. The source command file on the disk contains no uninitialized data.

Type 2 specifies the user stack area.

Text segments (type 3) specify the program's instructions.

Type 4 specifies the segment that contains the program's constants.

Type 5 identifies the segment(s) that contain data initialized within the command file.

Segment types 6 and 7 are provided for convenience in placing code and data in ROM. No mixed segment is allowed if the Z8000 is being operated as a split I and D space machine. A mixed type that is not protectable indicates that the code might need to store into the data items. A protectable mixed type can safely be put into ROM.

3.3 Relocation Data Section

The relocation data section of the file consists of a series of structures, each describing some item to be relocated. The relocation items are in the following form:

- The segment number (one byte): the ordinal number of the segment containing the item to be relocated. This number must be less than the number segments field in the header.
- The type of relocation to be performed (one byte). Table 3-3 lists the values for the relocation types.
- The location of item to be relocated (one unsigned word).
- An index to an entry in the symbol table, or the segment by which to relocate (one unsigned word).

Table 3-3. Relocation Type Values.

Value	Flag	Type of Relocation
1	OFF	Adjust a 16-bit offset value only
2	SSG	Adjust a short form (16-bit) segment plus offset
3	LSG	Adjust a long form (32-bit) segment plus offset

Table 3-3. (continued)

Value	Flag	Type of Relocation
5	XOF	Adjust a 16-bit offset, referenced by an external item
6	XSSG	Adjust a short segment, referenced by an external item
7	XLSG	Adjust a long segment, referenced by an external item

The first three relocation types are references between segments in the current file. For these cases, the relocation value is the segment number relevant to the file.

References to external symbols, types 5, 6, and 7, must be made using the symbol table.

3.4 Symbol Table

The symbol table defines the symbols referenced by the command file. A symbol table entry is a 12-byte structure divided into four fields that indicate the symbol's segment number, type, value, and name. Figure 3-3 depicts a symbol table entry.

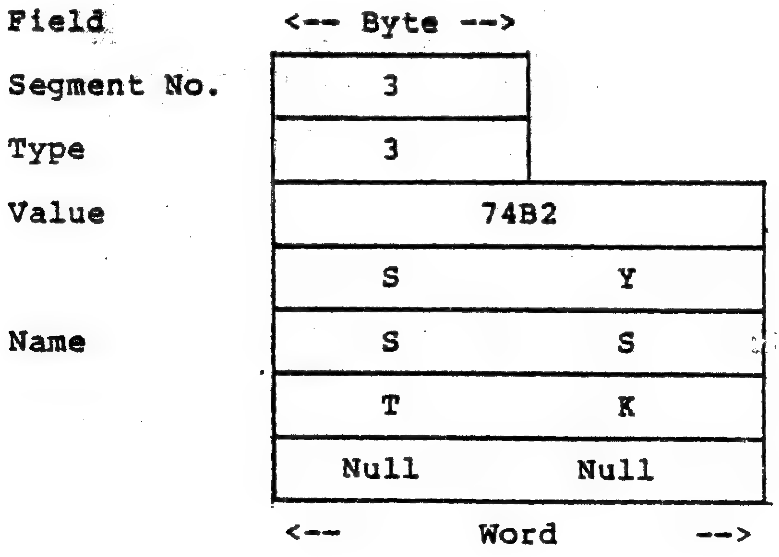


Figure 3-3. Symbol Table Entry

The first field of the symbol table entry, segment number, identifies the segment within the file that contains the symbol. A value of 255 indicates either an absolute or external reference.

Symbol type values are shown in Table 3-4.

The value field for undefined external references contains the amount of space to be allocated for the symbol. For other references, the value is the offset within the segment containing the symbol.

Table 3-4. Symbol Type Values

Value	Type
1	Local (Debugging only)
2	Undefined External
3	Global Definition
4	Segment Name

DDT for CP/M-8000 creates local symbol definitions for use during debugging sessions. These local symbols do not influence file linking.

Undefined external entries reference symbols that are contained in another module. However, a nonzero value field associated with such an entry will cause the linker to allocate space for the symbol.

Global entries reference symbols (either absolute or relocatable) defined in the current module.

The last type of symbol table entry, segment name (4), is used to point to a specific segment. When such an entry is made, the symbol name field is filled with a segment name that corresponds to the segment number field of the entry. The default segment names are

__text	First (or only) text segment
__text[x...]	Subsequent text segments
__data	First data segment
__data[x...]	Subsequent data segments
__bss	First uninitialized data (bss) segment
__bss[x...]	Subsequent bss segments

Segment name and number correspondences are accomplished by using the d option in the LDZ8K command line (see Section 6.2).

The last four words of the entry contain the symbol's name. This field is padded with null characters if the symbol's name is less than eight ASCII characters.

3.5 Printing the Symbol Table

CP/M-8000 provides two utilities with which you may print the symbol table of an object or command file, NMZ8K and XDUMP. XDUMP can also be used to print a file's header and segment information.

To invoke NMZ8K, use the following command line format:

```
NMZ8K filename
```

NMZ8K will only operate on object or command files. NMZ8K will not sort the symbols; it prints them in the order in which they appear in the file. The format of NMZ8K output is shown below:

```
symbols:

31E6     0    4     0 __text
31F2     1    4     0 __data
31FE     2    4     0 __bss
320A     0    3    50 __BDOS
3216   255    1     4 ARG2
```

The first column is the length of the symbol, offset from the previous symbol (12 bytes). The second column identifies the number of the segment (within the file) that contains the symbol. The third column is the symbol type. The fourth column represents the symbol's value (space to be allocated or offset within the segment). The last column of NMZ8K output shows the symbol's name or, for type 4 symbols, the name of the segment.

You can also use XDUMP to print the symbol table of an object or command file. Along with the symbol table, XDUMP prints the file's header and segment information. To invoke XDUMP for this purpose, include the -S option in the command line as follows:

```
XDUMP -S filename
```

XDUMP will print the header information, segment information, and symbol table for the object or command file identified by "filename" in the following format:

```
    magic = EE03 nseg = 4  init = 29870 reloc = 0  symb = 11928

      10 sg[0]:  sgno = 0  typ = 3   len = 27082
      14 sg[1]:  sgno = 1  typ = 4   len = 518
      18 sg[2]:  sgno = 2  typ = 5   len = 2270
      1C sg[3]:  sgno = 3  typ = 1   len = 7860

      segment 0     type is code
      segment 1     type is constant pool
      segment 2     type is initialized data
      segment 3     type is bss

      symbols:

        30    0     4     0 __text
        3C    0     3    10 cret
        48    0     3     0 csv
```

XDUMP prints the five fields of a file's header in the first row of its output. The next four rows represent the file's segment information array. Each of these rows indentifies a segment's assigned number, type value, and execution length.

The type values and contents of the file's segments are printed in the next four rows of XDUMP output. XDUMP formats the symbol table output just like NMZ8K: length, segment number, symbol type, symbol value, and symbol name. (See Section 7.3 for more complete information on the XDUMP utility.)

End of Section 3

Section 4
Basic Disk Operating System (BDOS Functions)

This section describes the operating system services available to transient programs through the Basic Disk Operating System (BDOS) module of CP/M-8000. The section begins with a description of BDOS functions and parameters, invocation procedures, and organization. Table 4-1 summarizes the CP/M-8000 BDOS functions.

Table 4-1. CP/M-8000 BDOS Functions

F#	Function	Type
0	System Reset	System/Program Control
1	Console Input	Character I/O, Console Operation
2	Console Output	Character I/O, Console Operation
3	Auxiliary Input*	Character I/O, Additional Serial I/O
4	Auxiliary Output*	Character I/O, Additional Serial I/O
5	List Output	Character I/O, Additional Serial I/O
6	Direct Console I/O	Character I/O, Console Operation
7	Get I/O Byte*	I/O Byte
8	Set I/O Byte*	I/O Byte
9	Print String	Character I/O, Console Operation
10	Read Console Buffer	Character I/O, Console Operation
11	Get Console Status	Character I/O, Console Operation
12	Return Version Number	System Control
13	Reset Disk System	Drive
14	Select Disk	Drive
15	Open File	File Access
16	Close File	File Access
17	Search for First	File Access
18	Search for Next	File Access
19	Delete File	File Access
20	Read Sequential	File Access
21	Write Sequential	File Access
22	Make File	File Access
23	Rename File	File Access
24	Return Login Vector	Drive
25	Return Current Disk	Drive
26	Set DMA Address	File Access
28	Write Protect Disk	Drive
29	Get Read-Only Vector	Drive
30	Set File Attributes	File Access
31	Get Disk Parameters	Drive
32	Set/Get User Code	System/Program Control
33	Read Random	File Access
34	Write Random	File Access
35	Compute File Size	File Access

* These functions must be implemented in the BIOS.

Table 4-1. (continued)

F#	Function	Type
36	Set Random Record	File Access
37	Reset Drive	Drive
40	Write Random With Zero Fill	File Access
46	Get Disk Free Space	Drive
47	Chain To Program	System/Program Control
48	Flush Buffers	System/Program Control
50	Direct BIOS Call	System/Program Control
59	Program Load	System/Program Control
61	Set Exception Vector	Exception
62	Set Supervisor State	Exception
63	Get/Set TPA Limits	Exception

4.1 BDOS Functions and Parameters

To invoke a BDOS function, you must specify one or more parameters. Each BDOS function is identified by a number, which is the first parameter you must specify. The function number is loaded in register R5. Some functions require a second parameter, which is loaded, depending on its size, in word register R7 or longword register RR6.

Byte parameters are passed as 16-bit words. The low order byte contains the data, and the high order byte should be zeroed. For example, the second parameter for the Console Output Function (2) is an ASCII character, a byte parameter. The character is loaded in the low order byte of R7.

Some BDOS functions return a value, passed in register R7. The hexadecimal value FFFF is returned in register R7 when you specify an invalid function number in your program.

Table 4-2 summarizes the registers used by CP/M-8000 BDOS functions.

Table 4-2. BDOS Parameter Summary

BDOS Parameter	Register
Function Number	R5
Word Parameter	R7
Longword Parameter	RR6
Return Value, if any	R7

4.1.1 Invoking BDOS Functions

After the parameters for a function are loaded in the appropriate registers, your program must specify a system call (SC) #2 instruction to access the BDOS and invoke the function. The following example illustrates the assembler syntax required to invoke the Console Output Function (2).

```
     ld      r5,#2    ;Loads the function number in R5

     ld      r7,#'U'  ;Loads the ASCII character U in R7

     sc      #2       ;Accesses the BDOS to invoke the function
```

In this example, the ASCII character uppercase U is output to the console. The assembler load instructions load register R5 with the number 2 for the BDOS Console Output Function and register R7 with the ASCII character uppercase U. A pair of single ('') or double ("") quotation marks must enclose an ASCII character. The SC #2 instruction invokes the BDOS Output Console Function, which echoes the character on the console's screen.

4.1.2 Organization Of BDOS Functions

The parameters of each BDOS function and its operation are described in the following sections. Each BDOS function is categorized according to the operation it performs. The BDOS function categories are as follows:

- File Access
- Drive Access
- Character I/O
- System/Program Control
- Exception

As you read the description of each function, notice that some functions require a physical address parameter designating the starting location of the Direct Memory Access (DMA) buffer or File Control Block (FCB). The DMA buffer is an area in memory where a 128-byte record resides before a disk write function and after a disk read operation. BDOS functions often use the DMA buffer to obtain or transfer data. The FCB, described in Section 4.2.1, is a 33- or 36-byte data structure used by BDOS file access functions.

4.2 File Access Functions

This section describes functions to create, delete, seek, read, and write files. File access functions are listed in Table 4-3.

Table 4-3. File Access Functions

Function	Function Number
Open File	15
Close File	16
Search For First	17
Search For Next	18
Delete File	19
Read Sequential	20
Write Sequential	21
Make File	22
Rename File	23
Set DMA Address	26
Read Random	33
Write Random	34
Compute File Size	35
Write Random With Zero Fill	40

4.2.1 A File Control Block (FCB)

Most of the file access functions in Table 4-3 require the address of a File Control Block (FCB). An FCB is a 33- or 36-byte data structure that provides file access information. The FCB can be 33 or 36 bytes when a file is accessed sequentially, but it must be 36 bytes when a file is accessed randomly. The last three bytes in the 36-byte FCB contain the random record number, which is used by random I/O functions and the Compute File Size Function (35). The starting location of an FCB must be an even-numbered address. The format of an FCB is shown in Figure 4-1. Table 4-4 contains definitions for each FCB field.

Field	dr	f1	f2	...	f8	t1	t2	t3	ex	s1	s2	rc	d0	...	dn	cr	r0	r1	r2
Byte	00	01	02	...	08	09	10	11	12	13	14	15	16	...	31	32	33	34	35

Figure 4-1. File Control Block (FCB) Format.

Table 4-4. File Control Block (FCB) Fields

FCB Field	Definition
dr	drive code (0 - 16) 0 => use default drive for file 1 => auto disk select drive A, 2 => auto disk select drive B, ... 16=> auto disk select drive P.
f1...f8	contain the filename in ASCII upper-case. High bit should equal 0 when the file is opened.
t1,t2,t3	contain the filetype in ASCII upper-case. The high bit should equal 0 when the file is opened. t1', t2', and t3' denote the high bit for the Set File Attributes Function (see Section 4.2.13). The following list shows which attributes are indicated when these bits are set and equal the value 1. t1' = 1 => Read-Only file t2' = 1 => SYS file t3' = 1 => Archive
ex	contains the current extent number, normally set to 00 by the user, but is in the range 0 - 31 (decimal) for file I/O
s1	reserved for internal system use.
s2	reserved for internal system use, set to zero for the Open (15), Make (22), and Search (17,18) file functions.
rc	record count field, reserved for system use
d0...dn	filled in by CP/M, reserved for system use
cr	current record to be read or written; for a sequential read or write file operation, the program normally sets this field to zero to access the first record in the file

Table 4-4. (continued)

FCB Field	Definition
r0,r1,r2	optional, contain random record number in the range 0-3FFFFH; bytes r0, r1, and r2 are a 24-bit value with the most significant byte r0 and the least significant byte r2. Random I/O functions use the random record number in this field.

For users of other versions of CP/M, note that CP/M-80..version 2.2, CP/M-68K, and CP/M-8000 perform directory operations in a reserved area of memory that does not affect the DMA buffer contents, except for the Search For First (17) and Search For Next (18) Functions. For these functions, the directory record is copied to the current DMA buffer.

4.2.2 File Processing Errors

When a program calls a BDOS function to process a file, an error condition can cause the BDOS to return one of five error messages to the console:

- CP/M Disk read error on drive x
- CP/M Disk write error on drive x
- CP/M Disk select error on drive x
- CP/M Disk change error on drive x
- CP/M Disk file error: ffffffff.ttt is read-only.

The variable x is a one-letter drive code that indicates the drive on which CP/M-8000 detects the error.

When CP/M-8000 detects one of these errors, the BDOS traps it. CP/M-8000 displays a message indicating the error and, depending on the error, allows you to abort the program, retry the operation, or continue processing. Each of these errors and their options are described below.

CP/M issues a CP/M Disk read or write error when the BDOS receives a hardware error from the BIOS. The BDOS specifies BIOS read and write sector commands when the BDOS executes file-related system functions. If the BIOS read or write routine detects a hardware error, the BIOS returns an error code to the BDOS that results in CP/M-8000 displaying a disk read or write error message at your console. In addition to the error message, CP/M-8000 also displays the following option message:

```
WARNING -- Do not attempt to change disk.
Do you want to Abort (A), Retry (R), or Continue with bad data (C)?
```

You may type one of the letters enclosed in parentheses and a RETURN in response to this message. Each of these options is described in Table 4-5.

Table 4-5. Read/Write Error Message Response Options

Option	Action
A	The A option or CTRL-C aborts the program and returns control to the CCP. CP/M-8000 returns the system prompt (>) preceded by the drive code.
R	The R option retries the operation that caused the error. For example, it rereads or rewrites the sector. If the operation succeeds, program execution continues as if no error occurred. If the operation fails, the error message and option message are redisplayed.
C	The C option ignores the error that occurred and continues program execution. The C option is not an appropriate response for all types of programs; program execution should not be continued in some cases. For example, if you are updating a data base and receive a read or write error but continue program execution, you can corrupt the index fields and the entire data base. For other programs, continuing program execution is recommended. For example, when you transfer a long text file and receive an error because one sector is bad, you may continue transferring the file. Review the file after it has been transferred. Using an editor, add the data that was not transferred owing to the bad sector.

Any response other than an A, R, C, or CTRL-C is invalid. The BDOS reissues the option message if you enter any other response.

CP/M-8000 displays the disk select error when you select a disk but receive an error due to one of the following conditions:

- You specified a disk drive not supported by the BIOS.
- The BDOS receives an error from the BIOS.
- You specified a disk drive outside the range A through P.

Before the BDOS issues a read or write function to the BIOS, the BDOS issues a disk select function to the BIOS. If the BIOS does not support the drive specified in the function, or if an error occurs, the BIOS returns an error to the BDOS. The BDOS then causes CP/M-8000 to display the disk select error on your console. If the error is caused by a BIOS error, CP/M-8000 returns the following option message:

```
WARNING -- Do not attempt to change disks

Do you want to Abort (A) or Retry (R)?
```

To select one of the options in the message, type one of the letters enclosed in parentheses. The A option terminates the program and returns control to the CCP. The R option tries to select the disk again. If the disk select function fails, CP/M-8000 redisplays the disk select error message and the option message.

However, if the error is caused because you specified a disk drive outside the range A through P, CP/M-8000 will display only the disk select error message. CP/M-8000 aborts the program and returns control to the CCP.

CP/M-8000 displays the CP/M disk change error message when the BDOS detects the disk in the drive is not the same disk that was logged in previously. Your program cannot recover from this error and will terminate. CP/M-8000 returns program control to the CCP.

You can log in a disk by accessing the disk or resetting the disk or disk system. The Select Disk Function (14) resets a disk. The Reset Disk System Function (13) resets the disk system. Files cannot be opened when your program invokes either of these functions.

You can use either a STAT command or the BDOS Set File Attributes Function (30) to set a file to read-only status. If you call the BDOS to write to a file that is set to read-only status, you will receive the CP/M disk file error and option messages shown below.

```
CP/M Disk File Error:  ffffffff.tt is read-only.

Do you want to: Change it to read/write (C), or Abort (A)?
```

The variable ffffffff.ttt in the error message denotes the filename and filetype. To select one of the options, type one of the letters enclosed in parentheses. Each option is described in Table 4-6.

Table 4-6. Disk File Error Response Options

Option	Action
C	Changes the status of this file from read-only to read-write and continues executing the program that was being processed when this error occurred.
A	Terminates execution of the program that was being processed and returns program control to the CCP. The status of the file remains read-only. If you enter a CTRL-C, it has the same effect as specifying this option.

CP/M-8000 reprompts with the option message if you enter any response other than those described above.

FUNCTION 15: OPEN FILE

```
Entry Parameters:
   Register   R5:  0FH
   Register  RR6:  FCB Address

Returned  Values:
   Register   R7:  Return Code

                   success:  00H - 03H
                   error:  FFH
```

The Open File Function matches the filename and filetype fields of the FCB specified in register RR6 with the corresponding fields of a directory entry for an existing file on the disk. When a match occurs, the BDOS sets the FCB extent (ex) field and the second system (S2) field to zero before the BDOS opens the file. Setting these one-byte fields to zero opens the file at the base extent, the first extent in the file. In CP/M-8000, files can be opened only at the base extent. In addition, the physical I/O mapping information, which allows access to the disk file through subsequent read and write operations, is copied to fields d0 through dn of the FCB. A file cannot be accessed until it has been successfully opened.

The Open File Function returns an integer value ranging from 00H through 03H in R7 when the open operation is successful. This function returns FFH in register R7 when it cannot locate the file.

The question mark (?) wildcard can be specified for the filename and filetype fields of the FCB referenced by register RR6. The ? wildcard has the value 3FH. For each position containing a ? wildcard, any character constitutes a match. For example, if the filename and filetype fields of the FCB referenced by RR6 contain only ? wildcards, the BDOS accesses the first directory entry. You should not create an FCB of all wildcards for this function because you cannot ensure which file this function will open using such a file specification.

Note that the current record field (cr) in the FCB must be set to zero by the program for the first record in the file to be accessed by subsequent sequential I/O functions. However, setting the current record field to zero is not required to open the file.

FUNCTION 16: CLOSE FILE

```
Entry Parameters:
   Register   R5:  10H
   Register  RR6:  FCB Address

Returned  Values:
   Register   R7:  Return Code

                   success:  00H - 03H
                     error:  FFH
```

The Close File Function performs the inverse of the Open File Function. When the FCB passed in RR6 has been previously opened by either an Open File (15) or Make File (22) Function, the close function updates the FCB in the disk directory. The process used to match the FCB with the directory entry is identical to the Open File Function (15). Close File returns an integer value ranging from 00H though 03H in R7 for a successful close operation. The value FFH is returned in R7 when the file cannot be found in the directory.

Closing the file is not required when only read functions access a file. However, when write functions access a file, it must be closed to update its disk directory entry.

FUNCTION 17: SEARCH FOR FIRST

```
Entry Parameters:
   Register   R5:  11H
   Register  RR6:  FCB Address

Returned  Values:
   Register   R7:  Return Code

                   success:  00H - 03H
                     error:  FFH
```

The Search For First Function scans the disk directory allocated to the current user number to match the filename and filetype of the FCB addressed in register RR6 with the filename and filetype of a directory entry. The value FFH is returned in register R7 when a matching directory entry cannot be found. An integer value ranging from 00H through 03H is returned in register R7 when a matching directory entry is found.

The directory record containing the matching entry is copied to the buffer at the current DMA address. Each directory record contains four directory entries of 32 bytes each. The integer value returned in R7 indexes the relative location of the matching directory entry within the directory record. For example, the value 01H indicates that the matching directory entry is the second one in the directory record in the buffer. The relative starting position of the directory entry within the buffer is computed by multiplying the value in R7 by 32 (decimal), which is equivalent to shifting the binary value of R7 left 5 bits.

All entries, including empty entries, for all user numbers on the default disk are searched when the drive (dr) field contains a ? wildcard. This function returns any matching entry, allocated or free, that belongs to any user number.

An allocated directory entry contains the filename and filetype of an existing file. A free entry is not assigned to an existing file. If the first byte of the directory entry is E5H, the entry is free. A free entry is not always empty. It may contain the filename and filetype of a deleted file because the directory entry for a deleted file is not zeroed.

FUNCTION 18: SEARCH FOR NEXT

```
Entry Parameters:
   Register   R5:  12H

Returned  Values:
   Register   R7:  Return Code

                   success:  00H - 03H
                     error:  FFH
```

The Search For Next Function scans the disk directory for an entry that matches the FCB and follows the last matched entry, found with this or the Search For First Function (17).

A program must invoke a Search For First Function (17) before invoking this function for the first time. Subsequent Search For Next Functions can follow, but they must be specified without other disk related BDOS functions intervening. Therefore, a Search For Next Function must follow either itself or a Search For First Function.

The Search For Next Function returns the value FFH in R7 when no more directory entries match.

FUNCTION 19: DELETE FILE

```
Entry Parameters:
   Register   R5:  13H
   Register  RR6:  FCB Address

Returned  Values:
   Register   R7:  Return Code

                   success:  00H
                     error:  FFH
```

The Delete File Function removes files and frees the directory entries for and space allocated to files that match the filename in the FCB pointed to by the address passed in RR6. The filename and filetype can contain wildcards, but the drive select code cannot be a wildcard as in the Search For First (17) and Search For Next (18) Functions. Use wildcards carefully. The Delete File Function will erase an entire disk directory if the asterisk (*) wildcard appears in both the FCB filename and filetype fields.

This function returns the value FFH in register R7 when it cannot find the referenced file. R7 contains the value 00H when the Delete File Function locates the file.

FUNCTION 20: READ SEQUENTIAL

```
Entry Parameters:
   Register   R5:  14H
   Register  RR6:  FCB Address

Returned  Values:
   Register   R7:  Return Code

                   success:  00H
                     error:  01H
```

The Read Sequential Function reads the next 128-byte record in a file. The FCB passed in register RR6 must have been opened by the Open File (15) or Make File Function (22) before this function is invoked. The program must set the current record field to zero following the open or make function to ensure the file is read from the first record in the file. After the file is opened, Read Sequential reads the 128-byte record specified by the current record field from the disk file to the current DMA buffer. The FCB current record (cr) and extent (ex) fields indicate the location of the record that is read. The current record field is automatically incremented to the next record in the extent after a read operation.

When the current record field overflows, the next logical extent is automatically opened and the current record field is reset to zero before the read operation is performed. After the first record in the new extent is read, the current record field contains the value 01H.

The value 00H is returned in register R7 when the read operation is successful. 01H is returned in R7 when the record being read contains no data. Normally, a "no data" situation is encountered at the end of a file. However, it can also occur when this function tries to read either a previously unwritten data block or a nonexistent extent. These situations usually occur with files created or appended with the BDOS Write Random Function (34).

FUNCTION 21: WRITE SEQUENTIAL

```
Entry Parameters:
   Register   R5:  15H
   Register  RR6:  FCB Address

Returned  Values:
   Register   R7:  Return Code

                   success:  00H
                     error:  01H or 02H
```

The Write Sequential Function writes a 128-byte record from the DMA buffer to the disk file whose FCB address is passed in register RR6. The FCB must be opened by either an Open File (15) or Make File (22) Function before your program invokes the Write Sequential Function. The DMA buffer record is written to the current record, as specified in the FCB current record (cr) field.

The FCB current record field is automatically incremented to the next record. When the current record field overflows, the next logical extent of the file is automatically opened and the current record field is reset to zero before the write operation. After the write operation, the current record field in the newly opened extent is set to 01H.

Records can be written to an existing file. However, newly written records can overlay existing records in the file because the current record field is usually set to zero after a file is opened or created. This action is performed to ensure a subsequent sequential I/O function accesses the first record in the file.

The value 00H is returned in register R7 when the write operation is successful. A nonzero value in register R7 indicates the write operation is unsuccessful due to one of the conditions described in Table 4-7.

Table 4-7. Unsuccessful Write Operation Return Codes

Value	Meaning
01	No available directory space - This condition occurs when the write command attempts to create a new extent that requires a new directory entry and no available directory entries exist on the selected disk drive.
02	No available data block - This condition is encountered when the write command attempts to allocate a new data block to the file and no unallocated data blocks exist on the selected disk drive.

FUNCTION 22: MAKE FILE

```
Entry Parameters:
   Register   R5:  16H
   Register  RR6:  FCB Address

Returned  Values:
   Register   R7:  Return Code

                   success:  00H - 03H
                     error:  FFH
```

The Make File Function creates and opens a new file on a specified or default disk. The address of the FCB for the file is passed in register RR6. You must ensure the FCB contains a legal filename that does not already exist in the referenced disk directory. The drive field (dr) in the FCB indicates the drive on which the directory resides. The disk directory is on the default drive when the FCB drive field contains a zero.

The BDOS creates the file and initializes the directory and the FCB in memory to indicate an empty file. The program must ensure that no duplicate filenames occur. Invoking the Delete File Function (19) prior to the Make File Function prevents the occurrence of duplicate filenames.

Register R7 contains an integer value in the range 00H through 03H when the function is successful. Register R7 contains the value FFH when a file cannot be created due to insufficient directory space.

FUNCTION 23: RENAME FILE

Entry Parameters:
Register R5: 17H
Register RR6: FCB Address

Returned Values:
Register R7: Return Code

success: 00H
error: FFH

The Rename File Function uses the FCB specified in register RR6 to change the filename and filetype of all directory entries for a file. As shown in Figure 4-2, the first 12 bytes of the FCB contain the file specification for the file to be renamed. Bytes 16 through 27 (d0 through d12) contain the new name of the file. The filenames and filetypes specified must be valid for CP/M. Wildcards can be specified in the filename and filetype fields. Use the Rename File Function carefully. The Rename File Function will rename all files on the drive if the wildcard "*" appears in both the filename and filetype fields.

The FCB drive field (dr) at byte position 0 selects the drive. The Rename File Function ignores the drive field at byte position 16 if it is specified for the new filename. Register R7 contains the value zero when the rename function is successful. It contains the value FFH when the first filename in the FCB cannot be found during the directory scan.

FCB byte position

0	1	2	3	4	5	6	7	8	9	10	11	...	16	17	18	19	20	21	22	23	...	27	...
dr	f1	f2	f3	f4	f5	f6	f7	f8	t1	t2	t3	...	d0	d1	d2	d3	d4	d5	d6	d7	...	d12	...

old file specification new file specification

Figure 4-2. FCB Format for Rename Function

Figure 4-2 uses horizontal ellipsis to indicate FCB fields that are not required for this function. Refer to Section 4.1.2 for a description of all FCB fields.

FUNCTION 26: SET DMA ADDRESS

```
Entry Parameters:
   Register   R5:  1AH
   Register  RR6:  DMA Physical Address

Returned  Values:
   Register   R7:  00H
```

The Set DMA Address Function sets the beginning physical address of the 128-byte DMA buffer. DMA is an acronym for Direct Memory Access. The DMA buffer is an area of memory that the disk controller can access directly to transfer data to and from the disk subsystem. Many computer systems use nonDMA access in which the data is transferred through programmed I/O operations. CP/M-8000 uses the DMA buffer to store the results of disk I/O regardless of whether the disk controller actually performs direct memory access or not. The DMA physical address in CP/M-8000 is the beginning physical address of a 128-byte data buffer, called the DMA buffer. The DMA buffer is the area in memory where a data record resides before a disk write operation and after a disk read operation. The DMA buffer can begin on an even or odd physical address.

Transient programs must convert a logical DMA address to the physical DMA address before calling the Set DMA Address Function. The entry parameter in register RR6 must be a physical address. Use the Map Address (_map_adr) System Call to convert a logical address to a physical address. Section 4.2 of the CP/M-8000 System Guide describes the _map_adr System Call in detail.

FUNCTION 30: SET FILE ATTRIBUTES

Entry Parameters:
Register R5: 1EH
Register RR6: FCB Address

Returned Values:
Register R7: Return Code

success: 00H
error: FFH

The Set File Attributes Function sets or resets file attributes supported by CP/M-8000 and user defined attributes for application programs. CP/M-8000 supports read-only, system, and archive attributes.

The high bit of each character in the ASCII filename (f1 through f8) and filetype (t1 through t3) fields in the FCB denotes whether attributes are set. When the high bit in any of these fields has the value 1, the attribute is set.

The address of the FCB is passed in register RR6. Wildcards cannot be specified in the filename and filetype fields.

This function searches the directory on the disk drive, specified in the FCB drive field (dr), for directory entries that match the FCB filename and filetype fields. All matching directory entries are updated with the attributes this function sets. This function returns a zero in register R7 when the attributes are set. If a matching entry cannot be found, register R7 contains FFH. Table 4-8 denotes the FCB fields and their attributes.

Table 4-8. File Attributes

Field	Attribute
f1 through f4	User-defined attributes for application programs.
f5 through f8	Reserved for future use by CP/M-8000.
t1	The Read-Only attribute indicates the file status is Read-Only. The BDOS does not allow write commands to operate on a file whose status is Read-Only. The BDOS does not permit a Read-Only file to be deleted.
t2	The System attribute indicates the file is a system file. Some built-in commands and system utilities differentiate between system and user files. For example, the DIRS command provides a directory of system files. The DIR command provides a directory of user files for the current user number. For details on these commands, refer to the <u>CP/M-8000 Operating System User's Guide</u>.
t3	The Archive attribute is reserved but not used by CP/M-8000. If set, it indicates that the file has been written to backup storage by a user-written archive program. To implement this facility, the archive program sets this attribute when it copies a file to backup storage; any programs updating or creating files reset this attribute. The archive program backs up only those files that have the Archive attribute attribute reset. Thus, an automatic backup facility restricted to modified files can be easily implemented.

The Open File (15) and Close File (16) Functions do not use the high bit in the filename and filetype fields when matching filenames. However, the high bits in these fields should equal zero when you open a file. Also, the Close File Function does not update the attributes in the directory entries when it closes a file.

FUNCTION 33: READ RANDOM

```
Entry Parameters:
   Register   R5:  21H
   Register  RR6:  FCB Address

Returned  Values:
   Register   R7:  Return Code

                   success:  00H
                     error:  01H, 03H
                             04H, 06H
```

The Read Random Function reads records randomly, rather than sequentially. The file must be opened with an Open File Function (15) or a Make File Function (22) before this function is invoked. The address of a 36-byte FCB is passed in register RR6. The FCB random record field denotes the record this function reads.

The random record field is a 24-bit field, with a value ranging from 00000H through 3FFFFH. This field spans bytes r0, r1, and r2 which are bytes 33 through 35 of the FCB. The most significant byte is first, r0, and the least significant byte, r2, is last. This byte sequence is consistent with the addressing conventions for the Z8000 and MC68000 microprocessors but differs from other versions of CP/M.

The random record number must be stored in the FCB random record field before the BDOS is called to read the record. After reading the record, register R7 contains either an error code (see Table 4-9), or the value 00H, which indicates the read operation was successful. When the read operation is successful, the current DMA buffer contains the randomly accessed record. The record number is not incremented. The FCB extent and current record fields are updated to correspond to the location of the random record that was read. A subsequent Read Sequential (20) or Write Sequential (21) Function starts from the record that was randomly accessed. Therefore, the randomly read record is reread when a program switches from randomly reading records to sequentially reading records. This is also true for the Write Random Functions (34, 40).

The last record written is rewritten if the program switches from randomly writing records to sequentially writing records with the Write Sequential Function (21). However, by incrementing the random record field following each Read Random Function (33) or Write Random Function (34, 40), a program can obtain the effect of sequential I/O operations.

Table 4-9 lists the numeric codes returned in register R7 following a random read operation.

Table 4-9. Read Random Function Return Codes

Code	Meaning
00	Success - returned in R7 when the Read Random Function succeeds.
01	Reading unwritten data - returned when a random read operation accesses a previously unwritten data block.
03	Cannot close current extent - returned when the BDOS cannot close the current extent prior to moving to the new extent containing the FCB random record number. This error can be caused by an overwritten FCB or a read random operation on an FCB that has not been opened.
04	Seek to unwritten extent - returned when a random read operation accesses a nonexistent extent. This error situation is equivalent to error 01.
06	Random record number out of range - returned when the value of the FCB random record field is greater than 3FFFFH.

FUNCTION 34: WRITE RANDOM

Entry Parameters:
Register R5: 22H
Register RR6: FCB Address

Returned Values:
Register R7: Return Code

success: 00H
error: 02H, 03H
05H, 06H

The Write Random Function writes a 128-byte record from the current DMA address to the disk file that matches the FCB referenced in register RR6. Before this function is invoked, the file must be opened with either the Open File Function (15) or the Make File Function (22).

This function requires a 36-byte FCB. The last three bytes of the FCB compose the random record field and contain the number of the record that is to be written to the file. You can append to an existing file by using the Compute File Size Function (35) to write the random record number to the FCB random record field.

For a new file, created with the Make File Function (22), you do not need to use the Compute File Size Function to write the first record in the file. Instead, specify the value 00H in the FCB random record field. The first record written to the newly created file is zero.

When an extent or data block must be allocated for the record, the Write Random Function allocates it before writing the record to the disk file. The random record number is not changed following a Write Random Function. Therefore, a new random record number must be written to the FCB random record field before each Write Random Function is invoked.

However, the logical extent number and current record field of the FCB are updated to correspond with the random record number that is written. Thus, a Read Sequential (20) or Write Sequential (21) Function that follows a Write Random Function either rereads or rewrites the record that was accessed by the Read or Write Random Function. To avoid overwriting the previously written record and simulate sequential write functions, increment the random record number after each Write Random Function.

After the random write function has completed its operation, register R7 contains either an error code (see Table 4-10), or 00H to indicate that the operation was successful.

Table 4-10. Write Random Function Return Codes

Code	Meaning
00	Success - returned when the Write Random Function succeeds without error.
02	No available data block - occurs when the Write Random Function attempts to allocate a new data block to the file, but the selected disk does not contain any unallocated data blocks.
03	Cannot close current extent - occurs when the BDOS cannot close the current extent before moving to the new extent that contains the record specified by the FCB random record field. This error can be caused by an overwritten FCB or a write random operation on an FCB that has not been opened.
05	No available directory space - occurs when the write function attempts to create a new extent but the selected disk drive has no available directory entries.
06	Random record number out of range - returned when the value of the FCB random record field is greater than 3FFFFH.

FUNCTION 35: COMPUTE FILE SIZE

Entry Parameters:			
Register	R5:	23H	
Register	RR6:	FCB Address	
Returned Values:			
Register	R7:	00H	
		success:	File Size written to FCB random Record Field
		error:	Zero written to FCB Random Record Field

The Compute File Size Function computes the size of a file and writes it to the random record fields of the 36-byte FCB whose address is passed in register RR6.

The FCB filename and filetype are used to scan the directory for an entry with a matching filename and filetype. If a match cannot be found, the value zero is written to the FCB random record field. When a match occurs, the virtual file size is written in the FCB random record field.

The virtual file size is the record number of the record following the end of the file. The virtual size of a file corresponds to the physical size when the file is written sequentially. However, the virtual file size may not equal the physical file size when the records in the file were created by random write functions. The Compute File Size Function computes the file size by adding the value 1 to the record number of last record in a file. However, for files that contain randomly written records, the record number of the last record does not necessarily indicate the number of records in a file. For example, the number of the last record in a sparse file does not denote the number of records in the file. Record numbers for sparse files are not usually sequential. Therefore, gaps can exist in the record numbering sequence. You can create sparse files with the Write Random Functions (34 and 40).

In addition to computing the file size, you can use this function to determine the end of an existing file. For example, when you append data to a file, this function writes the record number of the first unwritten record to the FCB random record field. When you use the Write Random (34) or the Write Random With Zero Fill (40) Function, your program appends data to the file more efficiently because the FCB already contains the appropriate record number.

FUNCTION 36: SET RANDOM RECORD

```
Entry Parameters:
   Register   R5:  24H
   Register  RR6:  FCB Address

Returned  Values:  Random Record
                   Field Set
```

The Set Random Record Function calculates the random record number of the current position in the file. The current position in the file is defined by the last operation performed on the file. Table 4-11 lists the current position relative to operations performed on the file.

Table 4-11. Current Position Definitions

Operation	Function	Current Position
Open file	Open File (15)	record 0
Create file	Make File (22)	record 0
Random read	Read Random (33)	last record read
Random write	Write Random (34) Write Random With Zero Fill (40)	last record written
Sequential read	Read Sequential (20)	record following the last record read
Sequential write	Write Sequential (21)	record following the last record written

This function writes the random record number in the random record field of the 36-byte FCB whose address your program passes in register RR6.

You can use this function to set the random record field of the next record your program accesses when it switches from accessing records sequentially to accessing them randomly. For example, your program sequentially reads or writes 128-byte data records to an arbitrary position in the file that is defined by your program. Your program then invokes this function to set the random record field in the FCB. The next random read or write operation that your program performs accesses the next record in the file.

Another application for the Set Random Record Function (36) is to create a key list from a file that you read sequentially. Your program sequentially reads and scans a file to extract the positions of key fields. After your program locates each key, it calls this function to compute the random record position for the record following the record containing the key. To obtain the random record number of the record containing the key, subtract one from the random record number that this function calculates. CP/M-8000 reads and writes 128-byte records. If your record size is also 128 bytes, your program can insert the record position minus one into a table with the key for later retrieval. By using the random record number stored in the table when your program performs a random read or write operation, your program locates the desired record more efficiently.

Note that if your data records are not equal to 128 bytes, your program must store the random record number and an offset into the physical record. For example, you must generalize this scheme for variable-length records. To find the starting position of key records, your program stores the buffer-relative position and the random record number of the records containing keys.

FUNCTION 40: WRITE RANDOM WITH ZERO FILL

```
Entry Parameters:
   Register   R5:  28H
   Register  RR6:  FCB Address

Returned  Values:
   Register   R7:  Return Code

                   success:  00H
                     error:  02H, 03H
                             05H, 06H
```

The Write Random With Zero Fill Function, like the Random Write Function (34), writes a 128-byte record from the current DMA buffer to the disk file. The address of a 36-byte FCB is passed in register RR6. The last three bytes contain the FCB random record field. This field specifies the record number of the record that this function writes to the file. Refer to Write Random Function (34) for details on the FCB and setting its random record field.

Like the Write Random Function, this function allocates a data block before writing the record when a block is not already allocated. However, in addition to allocating the data block, this function also initializes the block with zeroes before writing the record. If your program uses this function to write random records to files, it ensures that the contents of unwritten records in the block are predictable.

Upon completion of the random write function, register R5 contains either an error code (see Table 4-10), or 00H, which indicates that the operation was successful.

4.3 Drive Functions

This section describes drive functions that reset the disk system, select and write-protect disks, and return vectors. These functions are listed in Table 4-12.

Table 4-12. Drive Functions

Function	Function Number
Reset Disk System	13
Select Disk	14
Return Login Vector	24
Return Current Disk	25
Write Protect Disk	28
Get Read-Only Vector	29
Get Disk Parameters	31
Reset Drive	37
Get Disk Free Space	46

4.3.1 Reset Disk System Function

```
FUNCTION 13:  RESET DISK SYSTEM

    Entry Parameters:
       Register   R5:   0DH

    Returned  Values:
       Register   R7:   00H
```

The Reset Disk System Function restores the file system to a reset state. All disks are set to read-write (see the Write Protect Disk Function (28) and Get Read-Only Vector Function (29)), and all the disk drives are logged out. This function can be used by an application program that requires disk changes during operation. The Reset Drive Function (37) can also be used for this purpose. All files must be closed before your program invokes this function.

FUNCTION 14: SELECT DISK

```
Entry Parameters:
   Register   R5:  0EH
   Register   R7:  Disk Number

Returned  Values:
   Register   R7:  00H
```

The Select Disk Function designates the disk drive specified in register R7 as the default disk for subsequent file operations. The decimal numbers 0 through 15 correspond to drives A through P. For example, R7 contains a 0 for drive A, a 1 for drive B, and so forth through 15 for a full 16-drive system. In addition, the designated drive is logged-in if it is currently in the reset state. Logging in a drive places it in an on-line status which activates the drive's directory until the next cold start, or Reset Disk System (13) or Reset Drive (37) Function.

When the FCB drive code equals zero (dr = 0H), this function references the currently selected drive. However, when the FCB drive code value is between 1 and 16, this function references drives A through P.

If this function fails, CP/M-8000 returns a CP/M disk select error, as described in Section 4.2.2.

FUNCTION 24: RETURN LOGIN VECTOR

```
Entry Parameters:
   Register   R5:  18H

Returned  Values:
   Register   R7:  Login Vector
```

The Return Login Vector Function returns a 16-bit value that denotes the log-in status of the drives in register R7. The least significant bit corresponds to the first drive, A, and the high order bit corresponds to the sixteenth drive, labeled P. Each bit has a value of zero or one. The value zero indicates the drive is not on-line. The value one denotes the drive is on-line. When a drive is logged in, its bit in the log-in vector has a value of one. Explicitly or implicitly logging in a drive sets its bit in the log-in vector. The Select Disk Function (14) explicitly logs in a drive. File operations implicitly log in a drive when the FCB drive field (dr) contains a nonzero value.

FUNCTION 25: RETURN CURRENT DISK

Entry Parameters:
Register R5: 19H

Returned Values:
Register R7: Current Disk

The Return Current Disk Function returns the current default disk number in register R7. The disk numbers range from 0 through 15, which correspond to drives A through P. Note that this numbering convention differs from the FCB drive field, which specifies integers 1 through 16 for drives labeled A through P.

FUNCTION 28: WRITE PROTECT DISK

```
Entry Parameters:
   Register   R5:  1CH

Returned  Values:
   Register   R7:  00H
```

The Write Protect Disk Function provides temporary write protection for the currently selected disk. Any attempt to write to the disk, before the next cold start, warm start, disk system reset, or drive reset operation produces the message:

```
Disk change error on drive x
```

Your program terminates when this error occurs. Program control returns to the CCP.

FUNCTION 29: GET READ-ONLY VECTOR

```
Entry Parameters:
   Register   R5:  1DH

Returned  Values:
   Register   R7:  Read-Only
                   Vector Value
```

The Get Read-Only Vector Function returns a 16-bit vector in register R7. The vector denotes drives that have the temporary read-only bit set. Similar to the Return Login Vector Function (24), the least significant bit corresponds to drive A, and the most significant bit corresponds to drive P. The Read-Only bit is set either by an explicit call to the Write Protect Disk Function (28), or by the automatic software mechanisms within CP/M-8000 that detect changed disks.

FUNCTION 31: GET DISK PARAMETERS

```
Entry Parameters:
   Register    R5:  1FH
   Register   RR6:  CDPB Address

Returned  Values:
   Register    R7:  00H
             CDPB:  Contains DPB Values
```

The Get Disk Parameters Function writes a copy of the 16-byte BIOS Disk Parameter Block (DPB) for the current default disk (CDPB) at the address specified in register RR6. The entry parameter in RR6 can be a logical address.

The values in the CDPB can be extracted and used for display and space computation purposes. Normally, application programs do not use this function. Figure 4-3 illustrates the format of the DPB and CDPB. For more details on the BIOS DPB, refer to the CP/M-8000 Operating System System Guide.

SPT	BSH	BLM	EXM	RES	DSM	DRM	RES	CKS	OFF
16	8	8	8	8	16	16	16	16	16

Figure 4-3. DPB and CDBP

Table 4-13 lists the fields in the DPB and CDPB.

Table 4-13. Fields in the DPB and CDPB

Field	Description
SPT	Number of 128-byte logical sectors per track
BSH	Block shift factor
BLM	Block mask
EXM	Extent mask
RES	Reserved byte
DSM	Total number of blocks on the disk
DRM	Total number of directory entries on the disk
RES	Reserved for system use
CKS	Length (in bytes) of the checksum vector
OFF	Track offset to disk directory

FUNCTION 37: RESET DRIVE

```
Entry Parameters:
   Register   R5:  25H
   Register   R7:  Drive Vector

Returned  Values:
   Register   R7:  00H
```

The Reset Drive Function restores specified drives to the reset state. A reset drive is not logged-in and its status is read-write. Register R7 contains a 16-bit vector indicating the drives this function resets. The least significant bit corresponds to the first drive, A. The high order bit corresponds to the sixteenth drive, labeled P. Bit values of 1 indicate the drives this function resets.

To maintain compatibility with other Digital Research operating systems, this function returns the value zero in register R7.

FUNCTION 46: GET DISK FREE SPACE

```
Entry Parameters:
   Register   R5:  2EH
   Register   R7:  Drive Number

Returned  Values:
      DMA Buffer:  Free Sector Count
```

The Get Free Disk Space Function returns the free sector count in the first four bytes of the current DMA buffer. This function returns the free sector count as the number of free 128-byte sectors on the specified drive, which you pass as a drive number in register R7. CP/M-8000 assigns disk numbers sequentially from 0 through 15 (decimal). Each number corresponds to a drive in the range A through P. For example, the disk drive number for drive A is 0 and for drive B, the number is 1.

Note that these numbers do not correspond to those in the drive field of the FCB. The FCB drive field (dr) uses the numbers 1 through 16 (decimal) to designate drives.

4.4 Character I/O Functions

This section describes the functions that handle serial I/O for a physical device assigned to one of the four logical devices supported by CP/M-8000: CONSOLE, AUXIN, AUXOUT, and LIST.

BDOS character I/O functions read and write an individual ASCII character or character string to and from these devices, or report the devices' status. These functions are listed in Table 4-14.

Physical to logical device assignments are defined in the I/O Byte, described later in this section. The STAT command can be used to display and change current physical to logical device assignments. STAT is described in the CP/M-8000 User's Guide.

Table 4-14. Character I/O Functions

Function	Function Number
Console Input	1
Console Output	2
Direct Console I/O	6
Print String	9
Read Console Buffer	10
Get Console Status	11
Auxiliary Input	3
Auxiliary Output	4
List Output	5
Get I/O Byte	7
Set I/O Byte	8

FUNCTION 1: CONSOLE INPUT

```
Entry Parameters:
   Register   R5:  01H

Returned  Values:
   Register   R7:  ASCII Character
```

The Console Input Function reads the next character from the logical console device (CONSOLE) to register R7. Printable characters, along with carriage return, line feed, and backspace (CTRL-H), are echoed to the console. Tab characters (CTRL-I) are expanded into columns of eight characters. Other CONTROL characters, such as CTRL-C, are processed. All other nonprintable characters are returned in register R7 but not echoed to the console.

The BDOS does not return to the calling program until a character has been typed. Thus, execution of the program is suspended until a character is ready.

FUNCTION 2: CONSOLE OUTPUT

Entry Parameters:

Register	R5:	02H
Register	R7:	ASCII Character

Returned Values:

Register	R7:	00H

The Console Output Function sends the ASCII character from R7 to the logical console device. Tab characters expand into columns of eight characters. In addition, a check is made for stop scroll (CTRL-S), start scroll (CTRL-Q), and the printer switch (CTRL-P). This function also processes CTRL-C, which aborts the operation of the calling program and warm boots the system.

If the console is busy, execution of the calling program is suspended until the console accepts the character.

FUNCTION 6: DIRECT CONSOLE I/O

Entry Parameters:
Register R5: 06H
Register R7: 0FFH (input)
0FEH (status)
or
Character (output)

Returned Values:
Register R7: Character or Status

Direct Console I/O is supported under CP/M-8000 for those specialized applications that require character-by-character console input and output without the control character functions supported by CP/M-8000. This function bypasses all of CP/M-8000's normal CONTROL character functions such as CTRL-S, CTRL-Q, CTRL-P, and CTRL-C.

Upon entry to the Direct Console I/O Function, register R7 contains one of the values listed below.

Table 4-15. Direct Console I/O Function Values

Value	Meaning
FFH	denotes a CONSOLE input request
FEH	denotes a CONSOLE status request
ASCII character	output to CONSOLE where CONSOLE is the logical console device

When the input value is FFH, the Direct Console I/O Function calls the BIOS CONIN Function, which returns the next console input character in R7 but does not echo the character on the console screen. The BIOS CONIN function waits until it receives a character. Thus, execution of the calling program remains suspended until a character is ready.

When the input value is FEH, the Direct Console I/O Function returns the status of the console input in register R7. When register R7 contains the value zero, no console input exists. However, when the value in R7 is nonzero, console input is ready to be read by the BIOS CONIN Function.

When the input value in R7 is neither FEH nor FFH, the Direct Console I/O Function assumes that R7 contains a valid ASCII character, which is sent to the console.

FUNCTION 9: PRINT STRING

```
Entry Parameters:
   Register   R5:  09H
   Register  RR6:  String  Address

Returned  Values:
   Register   R7:  00H
```

The Print String Function sends the character string stored in memory at the address contained in register RR6 to the logical console device (CONSOLE). The print string is terminated by a dollar sign ($). Tabs are expanded as in the Console Output Function (2), and checks are made for stop scroll (CTRL-S), start scroll (CTRL-Q), and the printer switch (CTRL-P).

FUNCTION 10: READ CONSOLE BUFFER

Entry Parameters:	
Register R5:	0AH
Register RR6:	Buffer Address
Returned Values:	
Register R7:	00H
Register Buffer:	Character Count and Characters

The Read Buffer Function reads a line of edited console input from the logical console device (CONSOLE) to a buffer address passed in register RR6. Console input is terminated when the input buffer is filled, or a RETURN (CTRL-M) or line feed (CTRL-J) character is entered. The input buffer addressed by RR6 takes the form:

```
RR6:  +0 +1 +2 +3 +4 +5 +6 +7 +8    . . .    +n

      mx nc c1 c2 c3 c4 c5 c6 c7    . . .    ??
```

The variable mx is the maximum number of characters the buffer holds. The variable nc is the total number of characters placed in the buffer. Your program must set the mx value prior to invoking this function. The mx value can range in value from 1 through 255 (decimal). The characters entered from the keyboard follow the nc value. The value nc is returned to the buffer. It can range from 0 to the value of mx. If the nc value is less than the mx value, uninitialized characters follow the last character. Uninitialized characters are denoted as double question marks (??) in the representation of the input buffer shown above. A terminating RETURN or line feed character is not placed in the buffer and is not included in the total character count nc.

This function supports several editing control functions, which are briefly described in Table 4-16.

Table 4-16. Line Editing Controls

Keystroke	Result
RUB/DEL	removes and echoes the last character
CONTROL-C	reboots when it is the first character on a line
CONTROL-E	causes physical end-of-line
CONTROL-H	backspaces one character position
CONTROL-J	(line feed) terminates input line
CONTROL-M	(return) terminates input line
CONTROL-P	starts and stops the echoing of console output to the logical LIST device
CONTROL-Q	restarts console I/O after CTRL-S halts it
CONTROL-R	retypes the current line on the next line
CONTROL-S	halts console I/O and waits for CTRL-Q to restart it
CONTROL-U	echoes a pound sign (#) and advances the cursor to the next line, all previously input characters are ignored
CONTROL-X	backspaces to beginning of current line

Certain line editing controls return the cursor to its previous column position before invoking the Read Console Buffer Function. This convention makes your data input and line correction more legible.

FUNCTION 11: GET CONSOLE STATUS

```
Entry Parameters:
   Register   R5:  0BH

Returned  Values:
   Register   R7:  Console Status
```

The Get Console Status Function checks for the presence of a character typed at the logical console device (CONSOLE). If a character is ready, a nonzero value is returned in register R7; otherwise the value 00H is returned in R7.

FUNCTION 3: AUXILIARY INPUT

```
Entry Parameters:
   Register   R5:  03H

Returned  Values:
   Register   R7:  ASCII Character
```

The Auxiliary Input Function reads the next character from the auxiliary input device into register R7. The calling program remains suspended until the character is read. This function assumes the BIOS implements its Auxiliary Input Function. When more than one auxiliary input port exists, the BIOS should implement the I/O Byte Function. For details on the BIOS Auxiliary Input and I/O Byte Functions, refer to the CP/M-8000 System Guide.

FUNCTION 4: AUXILIARY OUTPUT

Entry Parameters:
Register R5: 04H
Register R7: ASCII Character

Returned Values:
Register R7: 00H

The Auxiliary Output Function sends a character from register R7 to the auxiliary output device. Execution of the calling program remains suspended until the hardware buffer receives the output character. This function assumes the BIOS implements its Auxiliary Output Function. When more than one auxiliary output port exists, the BIOS should implement the I/O Byte Function. For details on the BIOS Auxiliary Output and I/O Byte Functions, refer to the CP/M-8000 System Guide.

FUNCTION 5: LIST OUTPUT

```
Entry Parameters:
   Register   R5:  05H
   Register   R7:  ASCII Character

Returned  Values:
   Register   R7:  00H
```

The List Output function sends the ASCII character in register R7 to the logical list device (LIST).

FUNCTION 7: GET I/O BYTE

Entry Parameters:
Register R5: 07H

Returned Values:
Register R7: I/O Byte Value

The Get I/O Byte Function returns the current value of the I/O Byte in register R7. The I/O Byte is an 8-bit value that assigns physical devices, represented by 2-bit fields, to each of the logical devices: CONSOLE, AUXILIARY INPUT, AUXILIARY OUTPUT, and LIST. Figure 4-4 shows the format of the CP/M-8000 I/O Byte.

	most significant		least significant	
I/O Byte	LIST	AUXILIARY OUTPUT	AUXILIARY INPUT	CONSOLE
bits	7,6	5,4	3,2	1,0

Figure 4-4. I/O Byte

The value in each field of the I/O Byte ranges from 0-3 and defines the assigned source or destination of each logical device, as shown in Table 4-17.

Table 4-17. I/O Byte Field Definitions

CONSOLE field (bits 1,0)	
0	- console is assigned to the console printer (TTY:)
1	- console is assigned to the CRT device (CRT:)
2	- batch mode: use the AUXILIARY INPUT as the CONSOLE input, and the LIST device as the CONSOLE output (BAT:)
3	- user defined console device (UC1:)
AUXILIARY INPUT field (bits 3,2)	
0	- AUXILIARY INPUT is the Teletype device (TTY:)
1	- AUXILIARY INPUT is the high-speed reader device (PTR:)
2	- user defined reader # 1 (UR1:)
3	- user defined reader # 2 (UR2:)
AUXILIARY OUTPUT field (bits 5,4)	
0	- AUXILIARY OUTPUT is the Teletype device (TTY:)
1	- AUXILIARY OUTPUT is the high-speed punch device (PTP:)
2	- user defined punch # 1 (UP1:)
3	- user defined punch # 2 (UP2:)
LIST field (bits 7,6)	
0	- LIST is the Teletype device (TTY:)
1	- LIST is the CRT device (CRT:)
2	- LIST is the line printer device (LPT:)
3	- user defined list device (UL1:)

Please note that the Get I/O Byte Function is valid only if the BIOS implements its I/O Byte Function. The implementation of the BIOS I/O Byte Function is optional. PIP and STAT are the only CP/M-8000 utilities that use the I/O Byte. PIP accesses physical devices. STAT designates and displays the logical-to-physical device assignments. For details on implementing the I/O Byte Function, refer to the CP/M-8000 System Guide.

FUNCTION 8: SET I/O BYTE

```
Entry Parameters:
   Register   R5:  08H
   Register   R7:  I/O Byte Value

Returned  Values:
   Register   R7:  00H
```

The Set I/O Byte Function changes the system I/O Byte value to the value passed in register R7. This function allows programs to modify the current assignments for the four logical devices: CONSOLE, AUXILIARY INPUT, AUXILIARY OUTPUT, and LIST in the I/O Byte. This function is valid only if the BIOS implements its I/O Byte Function. Refer to the CP/M-8000 System Guide for details on implementing the I/O Byte Function. See the Get I/O Byte Function (7) for a description of the I/O Byte format and the possible values for its fields.

4.5 System/Program Control Functions

The system and program control functions described in this section warm boot the system, return the operating system version number, call the Basic I/O System (BIOS) functions, and terminate and load programs. These functions are listed in Table 4-18.

Table 4-18. System and Program Control Functions

Function	Function Number
System Reset	0
Return Version Number	12
Set/Get User Code	32
Chain to Program	47
Flush Buffers	48
Direct BIOS Call	50
Program Load	59

FUNCTION 0: SYSTEM RESET

Entry Parameters:
Register R5: 00H

Returned Values: Function Does Not Return to Calling Program

The System Reset Function terminates the calling program and returns control to the CCP command level.

FUNCTION 12: RETURN VERSION NUMBER

```
Entry Parameters:
   Register   R5:  0CH

Returned  Values:
   Register   R7:  Version Number
```

The Return Version Number Function provides information that allows version dependent programming. The one-word value 3022H is the version number returned in register R7 for version 1.1 of CP/M-8000.

Add the hexadecimal value 0200 to the version number when the system implements CP/NET.. For example, CP/M-80 Release 2.2 returns the version 0222H when the system implements CP/NET.

```
FUNCTION 32:  SET/GET USER CODE

Entry Parameters:
   Register   R5:  20H
   Register   R7:  FFH (get)
                      or
                   User Code
                     (set)

Returned  Values:
   Register   R7:  Current User
                   Number
```

An application program can change or obtain the currently active user number by calling the Set/Get User Code Function. If the value in register R7 is FFH, the value of the current user number is returned in register R7. The value ranges from 0 to 15 (decimal).

If register R7 contains a value in the range 0 through 15 (decimal), the current user number is changed to the value in register R7. When the program terminates and control returns to the CCP, the user number reverts to the BDOS default user number. The BDOS assumes the default is zero unless you specify the USER command to set an alternate default.

FUNCTION 47: CHAIN TO PROGRAM

Entry Parameters:
Register R5: 2FH

Returned Values:
Register R7: Function Does Not Return to Calling Program

The Chain to Program Function terminates the current program and executes the command line stored in the current DMA buffer. The format of the command line consists of a one-byte character count (N), the command line characters, and a null byte as shown in Figure 4-5. The character count contains the number of characters in the command line. The count must be no more than 126 characters. If an error occurs, you receive one of the CCP errors described in Appendix E.

N	Command Line (N characters)	0
1 byte	N bytes ≤ 126 bytes	1 byte

Figure 4-5. Command Line Format in the DMA Buffer

FUNCTION 48: FLUSH BUFFERS

Entry Parameters:
Register R5: 30H

Returned Values:
Register R7: Return Code

success: 00H
error: nonzero value

The Flush Buffers Function calls the BIOS Flush Buffers Function (21), which forces the system to write the contents of any unwritten or modified disk buffers to the appropriate disks. Control and editing applications use this function to ensure that data is periodically physically written to the appropriate disks. When the buffers are successfully flushed, this function returns the value 00H in register R7. However, if an error occurs, and this function does not complete successfully, this function returns a nonzero value in register R7.

FUNCTION 50: DIRECT BIOS CALL

```
Entry Parameters:
   Register   R5:  32H
   Register  RR6:  BPB Address (Physical)

Returned  Values:
   Register   R7:  BIOS Return Code
                        (if any)
```

Function 50 allows a program to call a BIOS function and transfers control through the BDOS to the BIOS. The RR6 register pair contains the address of the BIOS Parameter Block (BPB), a 5-word memory area containing two BIOS function parameters, P1 and P2, as shown in Figure 4-6. When a BIOS function returns a value, it is returned in register R7.

As with other BDOS functions, your program must specify a SC #2 instruction to invoke this BDOS function after the registers are loaded with the appropriate parameters. The starting location of the BPB must be an even-numbered address.

Direct BIOS calls such as Set DMA Address (12) and Get Address of Memory Region Table (18) require the use of CP/M-8000's Map Address (_map_addr) and Memory Copy (_mem_cpy) System Calls. _map_addr converts a logical address to a physical address. Logical addresses depend on the mode (system or normal) of the program using the address and on the space to be accessed (program or data). _mem_cpy copies a block of data from a specified source to a specified destination and can be used to copy the data stored at an address returned from a BIOS function call to the calling program's address space. See Section 4.2 of the CP/M-8000 System Guide for more information on the Map Address and Memory copy System Calls.

Field	Size
Function Number	1 word
Value P1	1 longword
Value P2	1 longword

Figure 4-6. BIOS Parameter Block (BPB)

In the Figure 4-6, the function number is a BIOS function number. (See Appendix A for a list of BIOS functions.) The two values, P1 and P2, are 32-bit BIOS parameters, which are passed in registers RR6 and RR8 before your program invokes the BIOS function.

For more details on BIOS functions, refer to the CP/M-8000 System Guide. Note that if the parameters are addresses, they are NOT mapped, so that the caller can specify addresses in any part of memory.

FUNCTION 59: PROGRAM LOAD

Entry Parameters:

Register	R5:	3BH
Register	RR6:	LPB Address (Physical)

Returned Values:

Register	R7:	Return Code
	success:	00H
	error:	01H - 03H

The Program Load Function loads an executable command file into memory. In addition to the function code, passed in register R5, your program must load register pair RR6 with the physical address of Load Parameter Block (LPB). Use the Map Address (_map_addr) system call to obtain the physical address of the LPB. _map_addr is described in Section 4.2 of the CP/M-8000 System Guide.

After a program is loaded, the BDOS loads register R7 with one of the return codes listed in Table 4-19.

Table 4-19. Program Load Function Return Codes

Code	Meaning
00	the function is successful
01	insufficient memory exists to load the file, or the header is bad
02	a read error occurs while the file is loaded in memory
03	bad relocation bits exist in the program file

The LPB describes the program and denotes the segment into which it is loaded. The format of the LPB is outlined in Figure 4-7. The Program Load Function reads the file header to complete the LPB before returning. The starting location of the LPB must be an even-numbered address.

Byte Offset	Content	Size
0H	address of FCB of successfully opened program file	1 longword
4H	segment in which to load non-segmented program	1 longword
8H	segment in which to load segmented program	1 longword
CH	address of base page (returned by BDOS)	1 longword
10H	default user stack pointer (returned by BDOS)	1 longword
14H	loader control flags	1 word

Figure 4-7. Format of the Load Parameter Block (LPB)

Before a program specifies the Program Load Function, the file must be opened with an Open File Function (15). The Open File Function is described in Section 4.2.

The loader control flags in the LPB select loader options as shown in Table 4-20.

Table 4-20. Load Parameter Block Options

Bit Number	Value	Meaning
0 (least significant byte)	0	Load program in non-segmented, non-split I & D TPA
	-1	Load program in segmented TPA
	1	Load program in non-segmented, split I & D TPA
1 - 15 (decimal)	0	Reserved, should be set to zero.

The CCP uses the Program Load Function to load a command file. The CCP places the base page address on the program's stack. The base page address is located at the address pointed to by register R15 (nonsegmented) or RR14 (segmented), the stack pointer. The program must return to the CCP by executing the BDOS Function 0 (Reset). The format of the base page is outlined in Appendix C.

The BDOS allocates memory for the base page within the segements specified in the LPB and returns the address of the allocated base page in the LPB. Locations 0000H - 0024H of the base page are initialized by the BDOS. Locations 0025H through 0037H are not initialized but are allocated and reserved by the BDOS. The CCP initializes the remaining base page values when it loads a program.

The BDOS allocates a user stack located just below the base page in the highest address of the TPA. The maximum size of the stack equals the address of the stack pointer minus the last address of the program plus 1. The value of the stack pointer is passed to the LPB by the BDOS.

For programs loaded by a transient program, rather than the CCP, refer to Section 2.2.3. Appendix B contains an example of a C program that illustrates how a transient program loads another program with the Program Load Function but without the CCP.

4.6 Exception Functions

This section describes the Set Exception Function (61), the Set Supervisor State Function (62), and the Get/Set TPA Limits Function (63) that set exceptions for error handling and other exception processing.

FUNCTION 61: SET EXCEPTION VECTOR

```
Entry Parameters:
   Register   R5:  3DH
   Register  RR6:  EPB Address

Returned  Values:
   Register   R7:  Return Code

                   success:  00H
                     error:  FFH
```

The Set Exception Vector Function allows a program to reset current exception vectors, set new exception vectors, and create exception handlers for the Z8000 microprocessor. Where possible, CP/M-8000 exception vector numbers match those used in CP/M-68K.

In addition to passing the function number in register R5, a program must pass the address of the Exception Parameter Block in register RR6. The EPB is a 10-byte data structure containing a one-word vector number and two longword vector values (see Figure 4-8). The EPB specifies the exception and the address of the new exception handler. The starting location of the EPB must be an even-numbered address.

Field	Size
Vector Number	1 word
New Defined Vector Value	1 longword
Old Vector Value Returned by BDOS	1 longword

Figure 4-8. Exception Parameter Block (EPB)

The vector number identifies the exception. The New Vector Value specifies the address of the new exception handler for the specified exception. The BDOS returns the value that the exception vector contained before this function was invoked in the Old Vector Value Field. The BDOS replaces the old vector value with the new vector value in its table of exception handlers and returns the address of the old exception handler to the old vector value in the EPB. After the BDOS sets the new exception vector, it passes the value 00H in register R7. However, if an error, such as a bad vector, occurs while the vector is being set, this function passes the value FFH in register R7. The bad vector error occurs when an invalid vector is specified for this function. Table 4-22 lists the valid exceptions that correspond to Z8000 microprocessor hardware.

Table 4-21. Valid Vectors and Exceptions

Vector	Exception
0	Nonmaskable Interrupt
1	EPU Trap
2	Segment Violation
8	Privilege Violation
32*	System Call 0 (debugger breakpoint)
36**	System Call 4
37**	System Call 5
38**	System Call 6
39**	System Call 7

* Vectors reserved for Resident System Extensions (RSX) implemented with the Get/Set TPA Limits Function (63).

** Recommended Trap vectors for applications.

When an exception occurs, before the BIOS passes control to the BDOS exception handler, it saves the registers on the system stack in a form suitable for use with the Control Transfer (_xfer) System Call provided by the BIOS (as described in Section 4.2 of the CP/M-8000 System Guide). The BIOS then calls the BDOS exception handler as a subroutine, in system, segmented mode. The BDOS exception handler restores the status to that of the program that was executing when the exception occurred (usually normal, nonsegmented mode), and places a copy of the context block on the appropriate (system or normal) stack. The exception handler must return control to the interrupted program using the _xfer System Call.

If an exception occurs for which no exception handler exists, the BDOS default exception handler returns an exception message to the logical console device (CONSOLE) before it aborts the program. The BDOS exception message format is defined below:

```
Exception nn at user address aaaaaaaa. Aborted.
```

where:

nn — is a hexadecimal number in the range 2H through 17H or 24H through 2FH that defines all exceptions excluding reset, hardware interrupts, and system Traps 0 through 3.

aaaaaaaa — is the address of the instruction following the one that caused the exception.

Except for exceptions handled by Resident System Extensions (RSXs), the BDOS reinitializes all vectors to the default exception handler when the BDOS System Reset Function (0) is invoked. Any exception vectors set by your program are reset after the BDOS warm boots the system. An RSX is a program that is not configured in the operating system but remains resident in memory after it is loaded. RSXs normally provide additional system functions. The Get/Set TPA Limits Function (63) allows you to create an area in the TPA in which one or more RSXs can reside.

FUNCTION 62: SET SUPERVISOR STATE

```
Entry Parameters:
   Register   R5:  3EH

Returned  Values:
   Register   R7:  00H
```

The Set Supervisor Function puts the calling program in supervisor (system) state. If the calling program is running on a segmented processor (Z8001 or Z8002), this function also puts the program into segmented mode. This function should not be used by novice programmers, and experienced programmers should be careful when invoking it.

The user stack is swapped when the program enters supervisor state. On return from this function, the stack pointer, register RR14, is the supervisor stack pointer and not the user stack pointer. Therefore, you cannot use register RR14 to reference the user stack. You must use the user stack pointer, which is accessible with the Load Control (LDCTL) instruction.

The supervisor stack is used by the BDOS and the BIOS. The size of this stack is unpredictable, and the percentage of the stack used by the system is dependent on the operation being performed and those previously performed. Therefore, you cannot predict how much of the stack is available for program operations. To avoid stack overflow and overwriting the system, you should not push more than a few dozen bytes onto the stack.

An alternate method of avoiding stack overflow is to switch to a private supervisor stack. You create the stack by loading into RR14 the address of an area in memory that you use as the supervisor stack. The address must be an even address. If you call BDOS and BIOS functions, your private supervisor stack should be 300 longwords more than the space required by the program. If your program exits supervisor mode, make sure your program restores the system stack pointer to its original value. The supervisor stack is reinitialized when the system warm boots.

FUNCTION 63: GET-SET TPA LIMITS

Entry Parameters:		
	Register R5:	3FH
	Register RR6:	TPAB Address
Returned Values:		
	Register R7:	00H
	Register TPAB:	Contains TPA Values

The Get/Set TPA Limits Function allows you to obtain or set the boundaries of the Transient Program Area (TPA). You must store the address of the Transient Program Area Block (TPAB) in register RR6. The TPAB is a 5-word data structure consisting of one word and two longwords. You create the TPAB in the TPA as illustrated in Figure 4-9.

Byte Offset	Field	Size
00H	Parameters	1 word
02H	Low TPA physical address	1 longword
06H	High TPA physical address + 1	1 longword

Figure 4-9. Transient Program Parameter Block

The value of the first two bits in the one-word Parameters Field determines whether this function gets or sets the TPA limits and which fields you supply. Figure 4-10 illustrates the format of the parameters field.

Parameters Field	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved bits (2-15) = 0															

bits: 1 0

values = 1/0 1/0

Figure 4-10. Parameters Field in TPAB

Bit Zero determines whether you get or set the TPA limits. When the value of bit zero is zero, the BDOS returns the current TPA boundaries in the Low and High Address fields of the TPAB. When the value of bit zero is one, the BDOS sets new TPA boundaries. The BDOS uses the values that you specify in the Low and High TPA physical address fields of the TPAB to set the new TPA boundaries. Use the Map Address (_map_adr) System Call to convert the Low and High TPA logical addresses to physical addresses. Section 4.2 of the CP/M-8000 System Guide describes the _map_addr System Call.

When you set the TPA boundaries, bit one determines whether the boundaries are temporary or permanent. When the value of bit one is zero, the TPA boundaries that you set are temporary; when the system warm boots, the previous TPA limits are restored. When the value of bit one is one, the TPA values that you set are permanent; they are not changed when the system warm boots.

Bits 2 to 15 contain zeroes and are reserved for future use.

Table 4-22.

Values For Bits 0 and 1 in the TPAB Parameters Field

Bit	Value	Explanation
0	0	Return boundaries of current TPA in TPAB Low and High Address Fields.
	1	Set new TPA boundaries with the values loaded in TPAB Low and High physical address fields.
1	0	Restore previous TPA values when the system warm boots.
	1	Permanently replace the TPA boundaries with the ones you specify in the Low and High TPAB physical address fields.

The following examples illustrate and explain values for bits zero and one.

Examples:

1) Get TPA Limits

1	0
0	0

This form of function 63 returns the boundaries of the current TPA in the Low and High address fields of the TPAB when the value of bit zero equals 0.

2) Temporarily Set TPA Limits

1	0
0	1

This form of the Get/Set TPA Limits Function temporarily sets the TPA boundaries to the boundaries that you supply in the Low and High physical address fields of the TPAB when bit zero equals 1 and bit one equals 0. The TPA boundaries are reset when the system warm boots.

3) Permanently Set TPA Limits

1	0
1	1

When bit zero equals one and bit one equals one, function 63 permanently sets the TPA boundaries to the values that you supply in the Low and High physical address fields of the TPAB. The TPA limits remain set until this function is called to reset the boundaries or the you cold boot the system.

End of Section 4

Section 5
ASZ8K Assembler

5.1 Assembler Operation

The CP/M-8000 Assembler, ASZ8K, assembles an assembly language source program for execution on the Z8000 microprocessor. It produces a relocatable object file and, optionally, a listing. The assembly language accepted by ASZ8K is identical to that of the Zilog Z8000 assembler as described in the Zilog Z8000 CPU User's Reference Manual. Appendix D of the present manual contains a summary of the instruction set. Exceptions and additions are described in Sections 5.5 and 5.6.

ASZ8K uses a predefinition file named ASZ8K.PD to reference Z8000 mnemonics against hexadecimal instruction codes. This ASCII file must be present in the current user number on the logged in drive in order to assemble a Z8000 source file with ASZ8K.

Use the XCON utility to convert the relocatable object files output by ASZ8K into the x.out format described in Section 3. XCON is described in Section 7.5.

Appendix E lists the error messages generated by ASZ8K.

5.2 Invoking the Assembler (ASZ8K)

Invoke ASZ8K by entering a command of the following form:

```
     ASZ8K [-o outfile] [-lux] file.8k{n|s}
```

Table 5-1 lists and describes the options associated with the ASZ8K command line.

Table 5-1. Assembler Options

Option	Meaning
-l or -L	Generates a listing file named file.lst if no -o outfile option and name are selected. The -x or -X option implies this as well.
-o or -O	Generates an output file with the outfile name. This command option also invokes the XCON utility to convert outfile to the x.out command file format.
-u or -U	Treat all undefined symbols in the assembly as global.
-x or -X	Generates an assembly and cross reference listing file named file.lst if no -o outfile option is selected. The -l or -L option implies this as well.
file.8kn or file.8ks	This is the only required parameter. It is the file specification of the assembly language source program to be assembled. With the .8kn extension, nonsegmented object code will be generated; the .8ks extension generates segmented object code.
outfile	If the -o or -O option is specified, an object file with the name outfile.rl will be generated instead of the default file.rl.

ASZ8K will display the following message if the command you enter to invoke the assembler contains a syntax error:

```
Usage:  asz8k [-o outfile] [-luxs] file.8k{n|s}
```

5.3 Assembly Language Directives

This section alphabetically lists and briefly describes the directives ASZ8K supports.

Table 5-2. Assembly Language Directives

```
Label    Directive    Operands
```

```
         .ABS
```

The .ABS directive causes the assembler to generate object code into the absolute section at the point most recently left off in that section. The absolute section is the default section for object code generation at the beginning of the assembly.

```
[label]  .ALIGN     expression
```

The .ALIGN directive forces alignment of the location counter in the current section. The expression in the operand field must be an integer between 0 and 16. Assuming the value of the expression is "a", the location counter is incremented, if necessary, until it is divisible by 2_a .

```
[label]  .BLOCK     expression
```

The .BLOCK directive reserves a block of contiguous memory locations. The expression in the operand field must be absolute. A block is reserved whose size in bytes is the expression value. The label, if present, is assigned the value of the location counter before the block is generated. A label can be used to reference the address of the first byte of the block.

```
[label]  .BYTE     {expression}
[label]  .BYTE      "string"
[label]  .BYTE      "string",0
```

The first .BYTE directive generates successive bytes of object code with specified values. The expressions are evaluated in order from left to right, and their values, truncated to 8 bits, are placed in successive bytes of object code. Label, if present, is assigned the value of the location counter before the first byte is generated.

Table 5-2. (continued)

<table>
<tr><th>Label</th><th>Directive</th><th>Operands</th></tr>
<tr><td colspan="3">The second .BYTE directive generates successive bytes of code whose values are specified by a string expression. Label, if present, is assigned the value of the location counter before the first byte is generated.
The third .BYTE directive generates successive bytes of code whose values are specified by a string expression, followed by a single byte whose value is zero. Label, if present, is assigned the value of the location counter before the first byte is generated.</td></tr>
<tr><td>label</td><td>.COMMON</td><td></td></tr>
<tr><td colspan="3">The .COMMON directive begins a new common section with a specified name. Label is required, and becomes the name of the new section.</td></tr>
<tr><td></td><td>.EJECT</td><td></td></tr>
<tr><td colspan="3">The .EJECT directive begins a new page in the listing file. The next line of the output listing will begin at the top of a new page. A form-feed character (ASCII code 0C hex or ^L) in the source file will produce the same effect, and is the preferred method of formatting listings.</td></tr>
<tr><td></td><td>.ELSE</td><td></td></tr>
<tr><td colspan="3">The .ELSE directive can appear only within a conditionally assembled portion of the source. This section begins with an .IF directive and ends with a matching .ENDIF directive. The effect of .ELSE is to reverse the effect of its matching .IF directive, or of a preceding .ELSE directive. If a preceding .IF directive or .ELSE directive caused assembly to be suppressed, the .ELSE causes it to resume. The reverse case is also true, and both are valid only until the matching .ENDIF or another .ELSE is encountered.</td></tr>
<tr><td></td><td>.END</td><td>[expression]</td></tr>
<tr><td colspan="3">The .END directive marks the end of a source file module, and optionally specifies a program starting address. If the expression is present in the operand field, its value will be used as the starting address when the program is loaded into memory and executed.</td></tr>
</table>

Table 5-2. (continued)

<table>
<tr><th>Label</th><th>Directive</th><th>Operands</th></tr>
<tr><td colspan="3">The .END directive must be the last statement of a source module. Any statements following .END are ignored by the assembler. If all source statements are exhausted without encountering an .END directive, the assembler will assume the presence of an .END directive (without operand) after the final source line.</td></tr>
<tr><td></td><td>.ENDIF</td><td></td></tr>
<tr><td colspan="3">The .ENDIF directive terminates a conditionally assembled portion of the source. The .ENDIF directive must always be matched by a previous .IF directive.</td></tr>
<tr><td></td><td>.ENDM</td><td></td></tr>
<tr><td colspan="3">The .ENDM directive terminates a macro definition. The .ENDM directive must always be matched by a previous .MACRO directive.</td></tr>
<tr><td></td><td>.ENDR</td><td></td></tr>
<tr><td colspan="3">The .ENDR directive terminates a repeated portion of the source. The .ENDR directive must always be matched by a previous .REPEAT directive.</td></tr>
<tr><td>[label]</td><td>.EQU</td><td>expression</td></tr>
<tr><td colspan="3">The .EQU directive permanently assigns a value and a type to an identifier. The label is given the value and type of the expression in the operand field. Any subsequent attempt to redefine the label will cause an error diagnostic to be printed.</td></tr>
<tr><td></td><td>.ERROR</td><td>string</td></tr>
<tr><td colspan="3">The .ERROR directive forces an error flag to be set, just as though a real assembly error had occurred. It is useful for detecting unexpected assembly time conditions, especially within macros. The first 1 - 3 characters of the specified string are placed into the error flags field of the assembly listing. The assembler's error count is appropriately incremented.</td></tr>
</table>

Table 5-2. (continued)

<table>
<tr><th>Label</th><th>Directive</th><th>Operands</th></tr>
<tr><td></td><td>.EXIT</td><td></td></tr>
<tr><td colspan="3">The .EXIT directive terminates a macro expansion or a repeat block before the assembler reaches the end. It is illegal outside of a macro expansion or a repeat block.

The current macro expansion or repeat block will be exited, and processing will continue with the first statement following the macro expansion or repeat block.</td></tr>
<tr><td></td><td>.GLOBAL</td><td>{identifier}</td></tr>
<tr><td colspan="3">The .GLOBAL directive declares one or more identifiers to be global (accessible to the linker).

Each of the identifiers in the operand field is declared to be global. Those which are defined within the current assembly will be available to other object modules at link time. Those which are not defined within the current assembly may be supplied by other object modules at link time.</td></tr>
<tr><td></td><td>.IF</td><td>expression</td></tr>
<tr><td colspan="3">Upon encountering an .IF directive, the assembler will evaluate the given expression. If the value of the expression is zero (false), the assembler will ignore subsequent statements until a matching .ELSE or .ENDIF directive is read. Statement processing will resume in the normal manner.

If the value of the expression is nonzero (true), the assembler will process subsequent statements normally. If a matching .ELSE directive is encountered, it will then ignore statements until it reads a matching .ENDIF directive.

Each .IF directive begins a conditionally assembled portion of the source which must be terminated by a matching .ENDIF directive. An .ELSE directive can optionally appear within that portion of the source.

Conditionally assembled portions of the source can be nested down to an arbitrary level.</td></tr>
</table>

Table 5-2. (continued)

<table>
<tr><th>Label</th><th>Directive</th><th>Operands</th></tr>
<tr><td></td><td>.INPUT</td><td>string</td></tr>
<tr><td colspan="3">The .INPUT directive causes the assembler to begin reading its source from a specified file.

The string in the operand field must be the name of a source file. The assembler will read from the specified file until it reaches the end. It will then resume reading from the original source file. The effect is as though the .INPUT directive was replaced by the contents of the specified file.</td></tr>
<tr><td>[label]</td><td>.LONG</td><td>{expression}</td></tr>
<tr><td colspan="3">The .LONG directive generates successive long words of object code, with specified values. Expressions are evaluated from left to right and their values are placed in successive long words of object code. The label, if present, is assigned the value of the location counter before the first long word is generated. This permits reference to the address of the first long word.</td></tr>
<tr><td>label</td><td>.MACRO</td><td></td></tr>
<tr><td colspan="3">The .MACRO directive begins the definition of a macro. Upon encountering a .MACRO directive, the assembler will define (or redefine) a macro whose name appears in the label field. The body of the macro is taken from subsequent source lines until a matching .ENDM directive is encountered.</td></tr>
<tr><td>[label]</td><td>.ORG</td><td>expression</td></tr>
<tr><td colspan="3">The absolute origin directive (.ORG) sets the location counter to the value of the expression. The label, if present, is assigned the new value of the location counter.

The assembler uses the value of location counter to assign absolute memory locations to subsequent statements. The expression cannot contain any forward, undefined, or external references.</td></tr>
</table>

Table 5-2. (continued)

<table>
<tr><th>Label</th><th>Directive</th><th>Operands</th></tr>
<tr><td></td><td>.REPEAT</td><td>expression</td></tr>
<tr><td colspan="3">The .REPEAT directive begins a portion of source which is to be repeated according to the number specified in the expression. If the expression is negative or zero, that portion of source is skipped entirely. A matching .ENDR directive terminates the .REPEAT directive. Repeated portions of source may be nested down to any manageable level.</td></tr>
<tr><td></td><td>.RESUME</td><td>identifier</td></tr>
<tr><td colspan="3">The .RESUME directive causes the assembler to resume using a previously declared section of source, referenced by the identifier in the operand field. Object code will be generated into the specified section at its most recently exited point.</td></tr>
<tr><td>label</td><td>.SECT</td><td></td></tr>
<tr><td colspan="3">The .SECT directive begins a new section with the name specified by the label.</td></tr>
<tr><td>label</td><td>.SET</td><td>expression</td></tr>
<tr><td colspan="3">The .SET directive assigns a value and a type to an identifier specified by the label. Both the value and type of the expression in the operand field are assigned to the label. The label can be redefined any number of times by subsequent .SET directives.</td></tr>
<tr><td></td><td>.SPACE</td><td>expression</td></tr>
<tr><td colspan="3">The .SPACE directive causes a specified number of blank lines to be produced in the listing file. If the expression specifies more lines than are left on the current listing page, the effect will be the same as the .EJECT directive. The source line containing .SPACE will not appear in the listing.</td></tr>
<tr><td></td><td>.STITLE</td><td>string</td></tr>
<tr><td colspan="3">The subtitle directive uses the string in the operand field to specify a new listing subtitle. .STITLE also begins a new page in the listing file. An .EJECT directive will be simulated, and .STITLE will not appear in the listing file.</td></tr>
</table>

Table 5-2. (continued)

<table>
<tr><th>Label</th><th>Directive</th><th>Operands</th></tr>
<tr><td></td><td>.TITLE</td><td>string</td></tr>
<tr><td colspan="3">The .TITLE directive uses the string in the operand field to specify a new listing title, and begin a new listing page. An .EJECT directive will be simulated, and .TITLE will not appear in the listing file.</td></tr>
<tr><td></td><td>.WARN</td><td>string</td></tr>
<tr><td colspan="3">The .WARN directive forces a nonfatal warning flag to be set in the assembler. This is useful for detecting unexpected assembly time conditions, especially within macros. The first 1 - 3 characters of the specified string are placed into the error flag field of the assembly listing. The assembler's warning count is appropriately incremented.</td></tr>
<tr><td>[label]</td><td>.WITHIN</td><td>expression</td></tr>
<tr><td colspan="3">The .WITHIN directive specifies the memory size that can be occupied by the current section. The expression for this directive is an integer from 0 through 32 representing an exponent to the base 2. This determines memory size for the current section, and begins at an address that is a multiple of $2^{expression}$.</td></tr>
<tr><td>[label]</td><td>.WORD</td><td>{expression}</td></tr>
<tr><td colspan="3">The .WORD directive generates successive words of object code with specified values. Expressions are evaluated from left to right, their values truncated to 16 bits, and placed in successive words of object code. The label, if present, is assigned the value of the location counter before the first word is generated.</td></tr>
</table>

5.4 Sample ASZ8K Commands

The following command assembles the source file TEST.8KN and produces the nonsegmented object file TEST.O. Error messages appear on the screen. Any undefined symbols are treated as global.

```
     A>ASZ8K -o TEST.O -u TEST.8KN
```

The command shown below assembles the source file SMPL.8KS and produces the segmented object file SMPL.O. Error codes and the assembly listing are contained in the file named SMPL.LST.

```
A>ASZ8K -o SMPL.O -l SMPL.8KS
```

Note that the XCON utility converts the object files produced by ASZ8K to the x.out format described in Section 3. XCON is described in Section 7.5.

5.5 Assembly Language Differences

The syntax differences between the ASZ8K assembly language and Zilog's assembly language are listed below.

1. To force the generation of a short form segmented address, prefix the address expression with a vertical bar (|). For example:

   ```
   ld    r2,loop
   ```

   will generate a long segmented address, while

   ```
   ld    r2,|loop
   ```

   will generate a short segmented address.

2. To form a hard segmented address at assembly time, use:

   ```
   [segment]offset
   ```

The ASZ8K assembler uses the following conventions:

1. ASZ8K accepts upper- and lower-case characters. You can specify instructions and directives in either case. However, labels and variables are case sensitive. For example, the labels "START" and "Start" are not equivalent.

2. ASCII string constants must be enclosed in double quotes:

   ```
   "ac14"
   ```

3. Registers can be referenced with the following mnemonics:

   ```
   r0-r15
   R0-R15
   rr0-rr14
   RR0-RR14
   rq0-rq12
   RQ0-RQ12
   ```

   Upper- and lower-case references are equivalent.

4. Comment lines must begin with a semicolon and end with a newline character. The comment field is included in the assembly listing file, but is otherwise ignored by the assembler.

5. Radix suffixes are as follows:

Radix	Suffix
2	B or b
8	O, Q, o, or q
10	D or d
16	H or h

In the last case, hexadecimal numbers, the letters A-F or a-f represent the numbers 10-15. Since a number must begin with a numeral, prefix a leading zero (0) to those hex numbers beginning with a letter.

5.6 Macro Descriptions

The ASZ8K assembler includes a feature that allows a group of statements to be defined as a macro. The use of macros consists of two distinct phases: macro definition and macro expansion.

Macro definition is initiated by the .MACRO directive, and is terminated by a matching .ENDM directive. No object code is generated at this point.

Macro expansion is the process of inserting a previously defined macro into the source, which is then processed normally by the assembler. Macro expansion is initiated by the appearance of a macro name in the operation field of a statement.

During macro expansion, variable arguments can be inserted into the generated source at previously defined locations. This permits a single macro to be used in many different instances of a general situation.

5.6.1 Macro Definition

ASZ8K macro definition takes the general form:

```
Label    Directive   Operands

label     .MACRO
                .
                .
                .
          .ENDM
```

The symbol in the label field of the .MACRO directive becomes the name of the macro. Source statements between the .MACRO and .ENDM directives become the body of the macro. Whenever the macro name appears in a subsequent operation field, the body of that macro will be inserted into the source.

5.6.2 Macro Expansion

The general form of a ASZ8K macro expansion is:

Label	Directive	Operands
[argument]	identifier	{argument}

The identifier in the operation field must be the name of a previously defined macro. Label and operand fields contain optional arguments that will be substituted into the body of the macro as it is expanded.

The arguments in the operand field are separated by blanks, tabs or commas. If an argument is to contain one of these separator characters, the entire argument can be enclosed in braces ({ }). The outermost set of enclosing braces will be removed from the argument before it is substituted into the macro body.

5.6.3 Macro Argument Substitution

During macro definition, the question mark (?) immediately followed by a digit is interpreted by the assembler as an argument substitution indicator. The digit following the question mark is used to select one of the arguments from the macro call. Arguments in the operand field are numbered sequentially from left to right, beginning with 1. The optional argument in the label field is argument 0. For example, "?2" would use the second argument in the operand field; "?0" selects the argument in the label field.

The number of arguments in a macro call need not match the number of arguments referenced in the macro definition. If there are too few arguments in the macro call, references to the missing arguments are replaced by empty strings. If there are too many arguments in the macro call, the extra arguments are available using the "??" argument specifier. See Section 5.6.4.

You can produce the question mark character in a macro expansion without argument substitution by preceding the question mark with a backslash (\?).

5.6.4 Referencing Extra Macro Arguments

The special argument specifier "??" can be used to reference extra arguments that might be present in a macro call. Extra arguments are identified as follows: each extra argument is enclosed in braces, these braced arguments are then combined into a list, separated by commas; the resulting string is used to replace each appearance of "??" in the macro definition.

When used with recursive macro calls, the ability to reference extra macro arguments allows you to write macros that can process a variable number of arguments.

5.6.5 Nesting Macro Definitions

A macro definition can contain another macro definition, which can contain a third macro, and so on, to any reasonable depth. An inner macro will not become defined until its outer macro is called and expanded.

5.6.6 Nesting Macro Calls

A macro can contain a call to another macro, which can contain a call to a third macro, and so on. A macro can also contain recursive calls to itself. The nesting of macro calls is limited only by practical considerations.

5.6.7 Macro Redefinition

If a previously defined macro name appears in the label field of a subsequent .MACRO directive, it will be redefined. The old macro definition is simply discarded and replaced by the new definition. Macros can be redefined any number of times.

End of Section 5

Section 6
LD8K Linker

6.1 Linker Operation

LD8K is the CP/M-8000 linker/loader that combines several ASZ8K assembled (object) programs into one executable file and creates absolute files from relocatable command files. It is capable of linking both segmented and nonsegmented Z8000 object modules. However, all of the modules linked must be of the same type. LD8K has facilities for searching libraries in the same format prepared by the archive utility, AR8K.

The error messages displayed by LD8K are listed in Appendix E.

6.2 Invoking the Linker (LD8K)

Invoke LD8K by entering a command of the following form:

```
     LD8K -w [-o outfile] [-inrstu] [-lxx] [-d] file ...
```

The default output file is named x.out. Arguments to LD8K are processed in their order of appearance in the command line. Libraries are searched exactly as they appear in a command. Only routines that resolve at least one external reference are loaded. Library ordering is important.

The type of the first object module encountered determines whether linking will be segmented or nonsegmented. Mixing of types is not permitted. Only entry points in modules with the correct type will be considered when archive files are searched.

Several options are available with LD8K. Except for library specification (-lxx), all options should appear before any file name. Table 6-1 lists and explains the LD8K command options.

Table 6-1. Linker Command Options

Option	Meaning
-d or -D	The next argument is taken to be a file name containing segment numbers and name correspondences. In this way, specific segment numbers can be assigned to named segments, or the loader can be forced to ignore specific segment numbers. There are two formats for the descriptor file. The first is a sequence of ASCII lines as follows: <number>[,<typechar>]*=<name>[,<name>]* or <number> The first form requires the named segments to be assigned the given number. The <typechar> should be one of the letters t, d, b, or c to indicate segment types: text, bss, data, or constant, respectively. If the type is not indicated, the type of the named segment will be used. If all of the named segments with the same name will not fit into 64KB, an error message will be issued. The named segments are assigned to the numbered segments according to the order in which the names appear in the control input line. The second form effectively reserves the segment number and does not permit the loader to use it. The second type of descriptor file is as follows: <type>=<number>[-<number>] [-<number>]* In this second form, the type is one of the names: text, bss, data, or cons, and the numbers are the valid numbers for this type of segment.

Table 6-1. (continued)

Option	Meaning

-d or -D
(cont.)
	Both forms cause the output file segments to appear in the order they are encountered in the descriptor file. In this way, the object order can be forced.
	text=11,9,7
	cons=10,12,8
	data=11-18
	bss=3-1
	The preceding example causes text to be put first in segment 11, then 9, then 7 (as each segment is filled in turn). It causes constants to be put into 10, 12, and 8. Data occupies segments 11 through 18, and the bss occupies segments 1 through 3.
	If segment types are mixed (text=5/data=5), both types of segments will be allocated first in segment five. Subsequent allocation will be separated however, if the file reads text=4,6/data=5,6.

-d"descriptor line"	This is a short form of the preceding control in which one line will suffice. The -d commands can be repeated, in which case they are processed in the order they are encountered.
-i or -I	Prepares the object file for split I and D space. That is, it starts the data addresses at zero. In the case of segmented loads, segment numbers can also be reused.
-lxx or -Lxx	Search the library name libxx.a. This may appear any place in the command line.

Table 6-1. (continued)

Option	Meaning
-n or -N	Mark the text read-only, so that it can be shared. This is meaningful only for nonsegmented loads. The data boundary is moved up to the next 8K boundary.
-o filename	The next argument is taken as the name of the object file instead of x.out.
-r or -R	Preserves relocation information in the output, even if all references are resolved.
-s or -S	Save space in the object file by stripping the output of the symbol table and relocation bits.
-t or -T	The next argument is a decimal number that sets the size of the stack segment.
-u or -U	Enters the next argument as an undefined reference so that loading can be accomplished solely from an archive file.

6.3 Sample Commands Invoking LD8K

The following command links the assembled file TEST.O into the file named TEST.8K and strips out the symbol table and relocation bits:

```
A>LD8K -s -o TEST.8K TEST.O
```

The assembled files A.O, B.O, and C.O are linked to produce the default output file X.OUT in the following command:

```
A>LD8K A.O B.O C.O
```

The command shown below produces an output file named TEST.8K by linking the assembled files TEST.O and TEST1.O.

```
     A>LD8K -o TEST.8K TEST.O TEST1.O
```

End of Section 6

Section 7
Programming Utilities

This section describes the five programming utilities supported by CP/M-8000: AR8K, DUMP, XDUMP, SIZEZ8K, and XCON. AR8K allows you to create and modify libraries. DUMP displays the contents of a file in hexadecimal and ASCII notation. XDUMP displays the header fields, segment information, initialized data, relocation information, and symbol table of an object or command file. SIZEZ8K displays the total size of a memory image command file and the size of each of its program segments. XCON converts ASZ8K object file output into the x.out format described in Section 3.

7.1 Archive Utility

AR8K, the archive utility create a library or replaces, adds, deletes, lists, or extracts object modules in an existing library. AR8K can be used on the C run-time library distributed with CP/M-8000 and documented in the C Language Programmer's Guide for CP/M-8000 for the Z8000 microprocessor.

7.1.1 AR8K Syntax

To invoke AR8K, specify the components of the following command line. Optional components are enclosed in square brackets [].

```
     AR8K DRQTX[V] ARCFILE [FILES...]
```

You can specify multiple object modules in a command line provided the command line does not exceed 127 bytes. The delimiter character between components consists of one or more spaces. Table 7-1 lists and explains the AR8K command line components.

Table 7-1. AR8K Command Line Components

Component	Meaning
AR8K	Invokes the Archive Utility. However, if you specify only the AR8K command, AR8K returns the following command line syntax and system prompt. A>**AR8K** usage: AR8K KEY ARCFILE [files. . .] A>

Table 7-1. (continued)

Component	Meaning
KEY	Indicates you must specify one of the following letters as an AR8K command: D, R, Q, T, X. Each of these one-letter commands is described in Table 7-2.
V	Indicates you can specify this one-letter option. The V option is described with the AR8K commands in Table 7-2.
ARCFILE	Indicates the library file specification.
FILES	Indicates object modules.

7.1.2 AR8K Operation

AR8K sequentially parses the command line once. AR8K searches for, inserts, replaces, or deletes object modules in the library in the sequence in which you specify them in the command line.

When AR8K processes a command, it creates a temporary work file called AR8KXXXX.TMP. AR8K uses AR8KXXXX.TMP when it processes AR8K commands. After the operation is complete, AR8K erases AR8KXXXX.TMP. However, depending on when an error occurs, AR8KXXX.TMP is not always erased. If this occurs, erase AR8KXXXX.TMP with the ERA command. Refer to Appendix E for error messages output by AR8K.

Table 7-2 lists and describes the AR8K commands. Examples in the table illustrate the affect of each command and its interaction with the V option.

Table 7-2. AR8K Commands and Options

Command	Option	Meaning
D		Delete one or more object modules (as specified in the command) from the library. You can specify the V option for this command.
	V	List the modules in the library being deleted by the D command:

Table 7-2. (continued)

Command	Option	Meaning
		A>AR8K DV RUSS.ARC ORC.O Deleting: orc.o A>
		The D command deletes the module ORC.O from the library RUSS.ARC. The V option displays "Deleting:" to indicate the action of the D command.
R		Create a library when the one specified in the command line does not exist. The R command can also be used to replace or add object modules to an existing library. Your command line must specify one or more object modules.
		You can replace more than one object module in the library by specifying module names in the command line. However, when the library contains more than one module with the same name, AR8K replaces only the first module it finds that matches the one specified in the command line. AR8K replaces modules already in the library only when your command line specifies the names of the existing modules before the names of the new modules to be added to the library. For example, if you specify the name of a module that you want replaced after the name of a module you are adding to the library, AR8K adds both modules to the end of the library.
		By default, the R command adds new modules to the end of the library. The R command adds an object module to a library if:
		• The object module does not already exist in the library. • The name of a module follows the name of a module that does not already exist in the library.

Table 7-2. (continued)

Command	Option	Meaning
		You may specify the V option with the R command to indicate the result of the operation performed on the library.
	V	List the object modules that the R command replaces or adds. **A>AR8K RV TOOLS.DBG NAIL.O WRENCH.O** Replacing: nail.o A> The R command replaces the object module NAIL.O and adds the module WRENCH.O to the library TOOLS.DBG. The V option lists the object modules being replaced by the R command.
Q		Quickly append the named module(s) to the end of the archive file specified in the command line. The Q command does not perform any checking to determine if the modules to be appended are already in the archive file.
	V	You can specify the V option with this command: **A>AR8K QV RUSS.ARC WORK.O MAIL.O** Appending: work.o mail.o The Q command appends the object modules WORK.O and MAIL.O to the library RUSS.ARC. The V option displays "Appending:" to indicate the action of the Q command.

Table 7-2. (continued)

Command	Option	Meaning
T		Print a table of contents or a list of specified modules in the library. If you do not specify object modules in the command line, the T command will print a table of contents for the entire library. The V option cannot be used with this command. **A>AR8K T LIBF.C** ftoa.o etoa.o atof.o etof.o A> The T command prints a table of contents in the library LIBF.C
X		Extract a copy of one or more object modules from a library and write them to the default disk. If no object modules are specified in the command line, the X command extracts a copy of each module in the library. The library is not modified.
V		List only those modules the X command extracts from the library. **A>AR8K XV LIBF.C ETOF.O FTOA.O** Extracting: etof.o ftoa.o

7.1.3 Errors

When AR8K encounters an error during an operation, the operation is not completed. The original library is not modified if the operation would have modified the library. Thus, no modules in the library are deleted, replaced, added, or extracted. Refer to Appendix E for error messages output by AR8K.

7.2 DUMP Utility

The DUMP Utility (DUMP) displays the contents of a CP/M file in both hexadecimal and ASCII notation. You can use DUMP to display any CP/M file regardless of the format of its contents (binary data, ASCII text, or executable file).

7.2.1 Invoking DUMP

Invoke DUMP by entering a command in the following format.

```
     DUMP [-shhhhhh] filenamel [>outfile]
```

Table 7-3 lists and describes the components of the DUMP command line.

Table 7-3. DUMP Command Line Components

Component	Meaning
-shhhhhh	This option allows you to specify a hexadecimal offset into the file to be dumped. When you specify this option, DUMP starts displaying the contents of the file from the byte-offset hhhhhh. By default, DUMP starts dumping the file's contents from the beginning of the file.
filename	The name of the file you want to dump.
>outfile	The greater than sign (>) followed by a filename or logical device optionally redirects DUMP output. You can enter any valid CP/M file specification, or one of the logical device names CON: (console) or LST: (list device). If you do not specify this optional parameter, DUMP sends its output to the console.

Note that DUMP will display the correct command format if you make an error while entering the DUMP command line:

```
     usage:  dump [-shhhhhh] file
```

DUMP displays its error messages at the console. See Appendix E for error messages outout by DUMP.

7.2.2 DUMP Output

DUMP sends its output to the console (or to a file or device, when specified), 8 words per line, in the following format:

```
rrrr oo (ffffff):  hhhh hhhh hhhh hhhh hhhh hhhh hhhh hhhh *aaaaaaaaaaaaaaaa*
```

Table 7-4. DUMP Output Components

Component	Meaning
rrrr	The record number (CP/M records are 128 bytes) of the current line of the display.
oo	The offset (in hex bytes) from the beginning of the CP/M record.
ffffff	The offset (in hex bytes) from the beginning of the file.
hhhh	The contents of the file displayed in hexadecimal.
aaaaaaaa	The contents of the file displayed as ASCII characters. If any character is not representable in ASCII, it is displayed as a period (.).

An example of DUMP output from a command file containing both binary and ASCII data is shown below.

```
A>DUMP dump.Z8K

0000 00 (000000):  601a 0000 1b34 0000 011d 0000 0e5e 0000 *`....4.......^..*
0000 10 (000010):  0000 0000 0000 0000 0900 ffff 6034 4320 *............`4C *
0000 20 (000020):  5275 6e74 696d 6520 436f 7079 7269 6768 *Run-time Copyright*
0000 30 (000030):  7420 3139 3832 2062 7920 4469 6769 7461 *t 1984 by Digital*
0000 40 (000040):  6c20 5265 7365 6172 6368 2056 3031 2c30 *l Research V01.0*
0000 50 (000050):  3320 206f 0004 2268 0018 2649 d3e8 001c *3  o.."h..&ISh..*

 . . . . (and so on) . . .
```

7.3 XDUMP Utility

The XDUMP utility displays the header fields, segment information, initialized data, relocation information, and symbol table of an object or command file. The format of CP/M-8000 command files is described in Section 3.

7.3.1 Invoking XDUMP

To invoke XDUMP, enter a command in the following format:

```
     XDUMP -D -R -S -X filename
```

Table 7-5 describes the components of the XDUMP command line.

Table 7-5. XDUMP Command Line Components

Component	Meaning
XDUMP	displays the header fields, segment information, initialized data, symbol table, and relocation information of the specified object or command file.
-D	displays the header fields, segment information, and initialized data portion of the specified object or command file.
-R	displays the header fields, segment information, and relocation information of the specified object or command file.
-S	displays the header fields, segment information, and symbol table of the specified object or command file.

Table 7-5. (continued)

Component	Meaning
-X	displays only the header fields and segment information of the specified command or object file.
filename	indicates the name of the object or command file you want to dump. XDUMP will ignore any characters entered beyond the filename component of the command line.

7.3.2 XDUMP Output

XDUMP sends its output to the console. XDUMP output is composed of five sections: header fields, segment information, segment contents, relocation information, and symbol table data.

The first row of XDUMP's output contains the five fields of the file header (see Section 3.1):

```
magic = EE02  nseg = 4  init = 9412  reloc = 4224  symb = 9528
```

XDUMP displays its segment information output in the following format:

```
10 sg[0]:  sgno = 255  typ = 3  len = 7996
14 sg[1]:  sgno = 255  typ = 4  len = 44
18 sg[2]:  sgno = 255  typ = 5  len = 1372
1C sg[3]:  sgno = 255  typ = 1  len = 204
```

The first column contains the hexadecimal offset from the beginning of the file and identifies the segment's logical number. The second column, sgno =, is the segment's preassigned number. The third column shows the segment's type value. The last column, len =, is a byte count (decimal) of the segment's execution length. For bss segments, len = is a byte count of the amount of space to be reserved for uninitialized data generated by the program during execution. (See Section 3.2.)

XDUMP also indicates the type of data contained in each segment:

```
segment 0      type is code
segment 1      type is constant pool
segment 2      type is initialized data
segment 3      type is bss
```

XDUMP uses the following format to display segment contents:

```
    oo    od    oh    hhhh hhhh hhhh hhhh hhhh hhhh hhhh hhhh

    20    0     0     e81e e822 e800 7c00 140e 0b00 bffe 1402
    30    16    10    0200 0100 7d44 7d55 2100 011d bb41 0020
    .                                                       .
    .                                                       .
    .                                                       .
```

Table 7-6 describes the components of this output.

Table 7-6. XDUMP Segment Contents Output Components

Component	Meaning
oo	The offset in hexadecimal bytes from the beginning of the file.
od	The offset in decimal bytes from the beginning of the segment.
oh	The offset in hexadecimal bytes from the beginning of the segment.
hhhh	The contents of the file displayed in hexadecimal words. Each line of output represents 8 words.

XDUMP displays its relocation data output in the following format:

```
    floc    sgn    flg    loc    bas

    24E4     0      5      68     19
    24EA     0      5      58     26
    .                             .
    .                             .
    .                             .
```

Where:

- floc is the hexadecimal offset from the beginning of the file.
- sgn is the ordinal number of the segment containing the item to be relocated.
- flg is the type of relocation to be performed.

- loc is the location of the item to be relocated.
- bas is the index to an entry in the symbol table or the segment by which to relocate.

See Section 3.3 for a description of command file relocation data.

The format of the symbol table data displayed by XDUMP is fully described in Section 3.5, Printing the Symbol Table.

7.4 SIZEZ8K Utility

The SIZEZ8K utility (SIZEZ8K) displays the sizes of each program segment within one or more command file and the total memory needed by each file. CP/M-8000 command files usually have a filetype of .Z8K or .REL.

The size of a command file returned by SIZEZ8K and the size of a command file returned by the STAT command are not equal. The file size returned by SIZEZ8K includes the size of the text, data, and bss program segments but does not include the size of the header, symbol table, and relocation data. For more details on the CP/M-8000 command file format, refer to Section 3. For details on the STAT command, refer to the CP/M-8000 User's Guide.

SIZEZ8K error messages are listed and described in Appendix E.

7.4.1 Invoking SIZEZ8K

Invoke SIZEZ8K by entering the following command line:

```
SIZEZ8K filename [filename2 filename3 ... ] [>outfile]
```

Table 7-7. SIZEZ8K Command Line Components

Component	Meaning
filename	The file specification of a command or object file whose size you want to determine.
filenamel filename2	Indicates one or more additional file specifications of files whose size you want to determine. SIZEZ8K can process multiple files, provided the command line does not exceed 128 bytes.

Table 7-7. (continued)

Component	Meaning
>outfile	Specifies the file to which SIZEZ8K sends its output. If you do not include an output file specification, SIZEZ8K sends its output to the console. You can specify a valid CP/M filename, or one of the logical device names CON: (console), or LST: (list device) for the SIZEZ8K output file.

7.4.2 SIZEZ8K Output

SIZEZ8K produces one output line for each input file you specify. When a non-segmented input file is specified, SIZEZ8K reports if it operates in split I and D mode. The output line is formatted as shown below:

```
filename: csize + dsize + bsize = tsize (hex)  stack size = ssize
```

Table 7-8 describes the components of the SIZEZ8K output line.

Table 7-8. SIZEZ8K Output Components

Component	Meaning
csize	The size, in decimal bytes, of the text segment of the file.
dsize	The size, in decimal bytes, of the data segment of the file.
bsize	The size, in decimal bytes, of the block storage segment (bss) of the file.
tsize	The total size, in decimal bytes, of the memory image occupied by the file. tsize is the sum of csize, dsize, and bsize.
hex	The same value as tsize, expressed in hexadecimal bytes.
ssize	The size of the stack required by the file.

For an explanation of the program segments of a command file, see Section 3, Command File Format.

7.4.3 SIZEZ8K Examples

This section contains three examples of SIZEZ8K command lines.

1. The SIZEZ8K command line specified in the following example returns the size of one command file and its program segments.

   ```
   A>SIZEZ8K stat.z8k

   STAT.Z8K:  7156 + 1358 + 60 = 8574  (217E) stack size = 0
           Split I/D program
   ```

   The program file STAT.Z8K contains a 7156-byte (decimal) text segment, a 1358-byte (decimal) data segment, and a 60-byte (decimal) bss. The total size of the program file is 8574 decimal bytes, which is the same as 217E hexadecimal bytes. The header in the STAT.Z8K file does not specify a minimum stack size. However, when CP/M-8000 loads a command file, it always reserves at least 256 bytes for the user stack. CP/M-8000 also creates a 256-byte base page. Therefore, to run STAT.Z8K, the minimum size of the TPA cannot be less than 9086 decimal bytes (8574 bytes for the program, 256 bytes for the stack, and 256 bytes for the base page). SIZEZ8K also reports that STAT.Z8K, a nonsegmented program, operates two segments, one for program instructions and one for data (called split I and D spaces).

2. The following SIZEZ8K command line returns the size of two program files and their program segments.

   ```
   A>SIZEZ8K sizez8k.z8k dump.z8k

   sizez8k.z8k:  7010 + 388 + 3706 = 11104 (2B60 )  stack size = 0
   dump.z8k:  6964 + 286 + 3678 = 10928 (2AB0 )  stack size = 0
   ```

   When you specify multiple file specifications in a SIZEZ8K command line, use a space to delimit each file specification.

3. SIZEZ8K will return an error message in response to this last example because the specified file is not a command or object file.

   ```
   A>SIZEZ8K clink.sub

   Not x.out format: clink.sub
   ```

CLINK.SUB is an ASCII file, not a command file. Files input to SIZEZ8K must be command or object files. Refer to Section 3 for the format of CP/M-8000 command files.

7.5 XCON Utility

The XCON utility converts ASZ8K relocatable object file output into the x.out format of CP/M-8000 command files. XCON output files are executable if they contain no unresolved references. See Section 3 for a description of the CP/M-8000 command file format. Note that ASZ8K will automatically invoke XCON when the assembler's -o option is included in the command line. Refer to Section 5.2 for a description of the ASZ8K -o option.

If XCON encounters any Z8001-only relocation entries during a conversion operation, it marks its output as a segmented file.

XCON error messages are described in Appendix E.

7.5.1 Invoking XCON

To invoke XCON, enter a command line in the following format:

```
XCON [-o outfile] [-s] file.obj
```

Table 7-9 describes the components of XCON command line.

Table 7-9. XCON Command Line Components

Component	Meaning
-o outfile	The -o outfile option directs XCON output to a specific file. You can specify any valid CP/M-8000 file specification. Note that CP/M-8000 uses a filetype of Z8K to recognize executable command files. XCON uses x.out as the default file specification when this option is not included in the command line.
-s	The -s option forces XCON to mark its output file as segmented regardless of the format of the input file.
file.obj	This is the specification of the object file you want XCON to convert to CP/M-8000's x.out command file format.

7.5.2 XCON Command Line Example

The following example of the XCON command line shows the ASZ8K object file named GENINT.OBJ being converted to the x.out format.

```
A>XCON -o intrpt.z8k genint.obj
```

This command line causes XCON to convert GENINT.OBJ into the x.out command file format. The -o option specifies that XCON output be placed in a file named INTRPT.Z8K.

End of Section 7

Section 8
DDT-Z8K

8.1 DDT-Z8K Operation

DDT-Z8K allows you to test and debug programs interactively in the CP/M-8000 environment. The description of DDT-Z8K operation in this section assumes you are familiar with the Z8000 Microprocessor, the assembler (ASZ8K) and the CP/M-8000 operating system.

Appendix E describes the error messages returned by DDT-Z8K.

8.1.1 Invoking DDT-Z8K

Invoke DDT-Z8K by entering one of the following commands:

```
     DDT

     DDT filename
```

The first command loads and executes DDT-Z8K. After displaying its sign-on message and hyphen prompt character (-), DDT-Z8K is ready to accept commands. The second command invokes DDT-Z8K and loads the program specified by filename for execution under the debugger.

8.1.2 DDT-Z8K Command Conventions

When DDT-Z8K is ready to accept a command, it prompts you with a hyphen (-). In response, you can type a command line of up to 64 characters. A command line must be terminated with a RETURN. Use the standard CP/M line-editing controls to correct typing errors while entering a command line. Table 4-16 summarizes the standard CP/M line-editing controls. DDT-Z8K does not process the command line until you enter a RETURN.

The first nonblank character of each command line determines the command action. Table 8-1 summarizes DDT-Z8K commands. Section 8.2 describes the DDT-Z8K commands in detail.

Table 8-1. DDT-Z8K Command Summary

Command	Action
A	assemble opcodes
B	set or display breakpoints
C	clear breakpoints
D	display memory in hexadecimal and ASCII
E	load program file for execution
F	fill memory block file with a constant
G	begin execution with optional breakpoints
H	sum/difference sum/difference
I	set up file control block and command tail
L	list memory using Z8000 mnemonics
M	move memory block
P	port data read or write
Q	quit or exit DDT
R	read file image into TPA
S	set memory to new values
T	trace program execution
U	untrace program monitoring
V	show memory layout of disk file read
W	write contents of memory block to disk
X	examine or modify CPU state
Y	set or clear flag status and control bits
$	calculate hexadecimal expressions; display number in decimal, hex and octal; display addresses/values for A, B, D, I, L, and $$ commands
$$	assign symbol values
?	display a summary of DDT-Z8K commands
RETURN	repeat previous command

One or more arguments can follow most command characters. These arguments may be hexadecimal values, filenames, or other information, depending on the command. You can separate arguments for any command with a comma or a space. Some commands can operate on byte, word, or longword data. The letter W for word or L for longword must be appended to the command character for commands that operate on multiple data lengths. You can enter a RETURN to repeat the previous command. Section 8.2 describes the command arguments in detail for each command.

8.1.3 Specifying Addresses

Most DDT-Z8K commands accept one or more addresses as operands. All addresses are entered as hexadecimal numbers of up to eight hexadecimal digits. DDT-Z8K will right justify and zero pad from the left any address you enter with less than eight hexadecimal digits.

8.1.4 Terminating DDT-Z8K

Terminate DDT-Z8K by typing a CTRL-C (^C) or the Q command in response to the hyphen prompt. DDT will return control to the CCP.

8.1.5 DDT-Z8K Operation with Interrupts

DDT-Z8K operates with interrupts enabled or disabled, and preserves the interrupt state of the program being executed under DDT-Z8K. When DDT-Z8K has control of the CPU, either when it is initially invoked, or when it regains control from the program being tested, the condition of the interrupt mask is the same as it was when DDT-Z8K was invoked, except for a few critical regions where interrupts are disabled. While the program being tested has control of the CPU, the user's CPU state, which can be displayed with the X command, determines the state of the interrupt mask.

Note that DDT-Z8K uses the Vectored Interrupt and Privileged Instruction Trap exceptions. Therefore, programs debugged under test should not use these exceptions.

8.2 DDT-Z8K Commands

This section defines DDT-Z8K commands and their arguments. DDT-Z8K commands give you control of program execution and allow you to display and modify system memory and the CPU state.

8.2.1 The A (Assemble) Command

The A command causes DDT-Z8K to assemble the opcodes you enter from the terminal. You enter a single instruction opcode mnemonic and operands as arguments of the A command. You may also enter the A command without opcode arguments. This will cause DDT-Z8K to query the terminal for instruction opcodes. DDT-Z8K will continue to query the terminal for instruction opcodes until you enter a command line consisting of only a carriage return.

The forms of the A command are

```
A [optional address]
A [optional address],[instruction opcode mnemonic],[operand],...
```

If you enter the A command without specifying the address field, DDT-Z8K will assemble instructions at the current location of the A command pointer. If you enter the A command without specifying the instruction opcode mnemonic and operands, DDT-Z8K will query the terminal for instructions to assemble. DDT-Z8K will stop querying the terminal for instructions when you enter a carriage return alone in response to a query.

8.2.2 The B (Breakpoint) Command

The B command lets you set and display breakpoints in the program you are debugging. The forms of the B command are

```
        B
        B[address 1],[address 2],...[address 10]
```

The first form of the B command displays all set breakpoints. The second form sets breakpoints at up to ten addresses. Use the C command to clear breakpoints.

8.2.3 The C (Clear) Command

The C command allows you to clear breakpoints in the program you are debugging with DDT-Z8K. The forms of the C command are

```
        C
        C[address 1],[address 2],...[address 10]
```

The first form of the C command clears all set breakpoints. The second form clears breakpoints at up to ten addresses. Use the B command to set breakpoints.

8.2.4 The D (Display) Command

The D command displays the contents of memory as 8-bit, 16-bit, or 32-bit hexadecimal values and in ASCII code. The forms of the Display command are

```
     D
     Ds
     Ds,f
     DW
     DWs
     DWs,f
     DL
     DLS
     DLS,f
```

where s is the starting address, and f is the last address that DDT-Z8K displays.

Memory is displayed on one or more lines. Each line shows the values of up to 16 memory locations. For the first three forms, the display line appears as follows:

```
     aaaaaaaa bb bb ... bb cc ... cc
```

where aaaaaaaa is the address of the data being displayed, bb is the contents of the memory locations in hexadecimal, and cc represents the contents of memory in ASCII. Any nongraphic ASCII characters are represented by periods.

In response to the Ds form of the D command, shown above, DDT-Z8K displays 12 lines that start from the current address. Form Ds,f displays the memory block between locations s and f. Forms DW, DWs, and DWs,f are identical to D, Ds, and Ds,f except that the contents of memory are displayed as 16-bit values, as shown below:

```
     aaaaaaaa  wwww  wwww  ...  wwww  cccc  ...  cc
```

Forms DL, DLs, and DLs,f are identical to D, Ds, and Ds,f except that the contents of memory are displayed as 32-bit or longword values, as shown below:

```
     aaaaaaaa llllllll llllllll ... llllllll cccccccc ...
```

You can cancel the D command display at any time by typing any character at the console.

8.2.5 The E (Load for Execution) Command

The E command loads a file in memory so that a subsequent G, T or U command can begin program execution. The syntax for the E command is

```
     Efile
```

file represents the file specification of the file to be loaded.

An E command reuses memory used by any previous E command. This means that only one file at a time can be loaded for execution.

When the load is complete, DDT-Z8K displays the file's magic number, the starting and ending addresses of each segment in the file, and the command tail. Use the V command to display this information later.

If the file does not exist or cannot be successfully loaded in the available memory, DDT-Z8K displays the following error message:

```
     error #80         inload:  can't open inload file
```

Appendix E describes the other error messages returned by DDT-Z8K.

8.2.6 The F (Fill) Command

The F command fills an area of memory with a byte, word, or longword constant. The forms of this command are

```
     Fs,f,b
     FWs,f,w
     FLs,f,l
```

s is the starting address of the block to be filled, and f is the address of the final byte of the block within the segment specified in s.

In response to the first form, DDT-Z8K stores the 8-bit value b in locations s through f. In the second form, the 16-bit value w is stored in locations s through f in standard form--the high 8 bits are first, followed by the low 8 bits. In the third form, the 32-bit value l is stored in locations s through f, with the most significant byte first.

If you do not supply all three arguments (s, f, and b, w, or l) in the Fill command line, DDT-Z8K displays the following error message:

```
     error #63         dbg:     need 3 args
```

DDT-Z8K displays the error message below when the value you specify for s is greater than or equal to the value you specify for f.

```
     error #66         dbg:     arg1 >= arg2
```

8.2.7 The G (Go) Command

The G command transfers control to the program being tested, and optionally sets one to ten breakpoints. The Go command lines are

```
     G
     G,b1,...b10
     Gs
     Gs,b1,...b10
```

s is the address where you want the loaded program to begin executing, and b1 through b10 are the addresses of the breakpoints you want to set.

In the first two forms of the G command, no starting address is specified. The program starts executing at the address specified by the program counter (PC). The first form transfers control to the program without setting any breakpoints. The second form sets breakpoints before passing control to the program. The next two forms are analogous to the first two, except that the PC is first set to s.

Once control has been transferred to the program under test, that program executes in real time until a breakpoint is encountered. DDT-Z8K then regains control, clears all breakpoints, and displays the CPU state in the same form as the X command. When a breakpoint returns control to DDT-Z8K, the instruction at the breakpoint address has not yet been executed. To set a breakpoint at the same address, you must first specify a T or U command.

8.2.8 The H (Hexadecimal Math) Command

The H command computes the sum and difference of two 32-bit values. The form of the H command is

```
Ha,b
```

a and b are the values whose sum and difference are computed. DDT-Z8K displays the sum (ssssssss) and the difference (dddddddd) truncated to 16 bits on the next line:

```
ssssssss dddddddd
```

8.2.9 The I (Input Command Tail) Command

The I command prepares a file control block (FCB) and command tail buffer in DDT-Z8K's base page, and copies the information in the base page of the last file loaded with the E command. The form of this command is

```
Icommand tail
```

command tail is a character string, which usually contains one or more filenames.

The first filename is parsed into the default file control block at 005CH. The optional second filename, if specified, is parsed into the second default file control block beginning at 0038H. The characters in the command tail are also copied to the default command buffer at 0080H. The length of the command tail is stored at 0080H, followed by the character string terminated with a binary zero.

If a file has been loaded with the E command, DDT-Z8K copies the file control block and command buffer from the base page of DDT-Z8K to the base page of the loaded program.

8.2.10 The L (List) Command

The L command lists the contents of memory in assembly language. The forms of the List command are

```
L
Ls
Ls,f
```

The starting address is indicated by s, and f is the last address in the list.

The first form lists 12 lines of disassembled machine code, starting from the current address. The second form sets the list address to s and then lists 12 lines of code. The last form lists disassembled code from s through f. In all three cases, the list address is set to the next unlisted location in preparation for a subsequent L command. When DDT-Z8K regains control from a program being tested (see G, T and U commands), the list address is set to the address in the program counter (PC).

You can cancel long displays by typing any key during the list process. To halt the display temporarily, enter a Ctrl-S (^S). Type Ctrl-Q to restart the display after Ctrl-S has halted it.

In general, the syntax of the assembly language statements produced by the L command is standard Z8000 assembly language, as described in the Z8000 CPU User's Reference Manual, Prentice-Hall, 1982. Three minor exceptions are

- DDT-Z8K prints all numeric values in hexadecimal.
- DDT-Z8K uses lowercase mnemonics.
- DDT-Z8K assumes word operations unless a byte or longword specification is explicitly stated.

8.2.11 The M (Move) Command

The M command moves a block of data values from one area of memory to another. The form of the Move command is

```
Ms,f,d
```

s is the starting address of the block to be moved, f is the address of the final byte to be moved, and d is the address of the first byte of the area to receive the data.

Note that if the value represented by d is between s and f, some of the block being moved will be overwritten before it is moved, because data is transferred starting from location s.

8.2.12 The P (Port Read/Write) Command

The P command allows you to read data from or write data to an I/O port. The P command has four forms:

```
P port#
PW port#
P port #[,byte 1][,byte 2]...
PW port #[,word 1][,word 2]...
```

The first form of the P command causes DDT-Z8K to read and display the first eight bits from the port indicated by port#. The second form reads and displays the word at the indicated port. The third form of the P command writes the argument bytes to the specified port. The fourth form writes the argument words to the specified port.

Table 8-2 shows the Z8000 port numbers and their assignments for the Olivetti.. M20.

Table 8-2. Z8000 Port Assignments for the Olivetti M20

Port Number	Assignment
81	Output to Parallel Port
83	Input from Parallel Port
85	Parallel Port I/O Bits
87	Parallel Port Control Byte
C1	Serial Port I/O
C3	Serial Port Status/Control

8.2.13 The R (Read) Command

The R command reads a file to a contiguous block in memory. Its format is

```
Rfile[,offset]
```

file is the name and filetype of the file to be read.

DDT-Z8K reads the file in memory and displays the starting and next available addresses of the block of memory occupied by the file. A Value (V) command can redisplay the information at a later time. The default breakpoint and list pointers (for subsequent B and L commands) are set to the start of the file's code segment.

You can include an optional offset in the R command line. The offset must be specified in hexadecimal; DDT-Z8K loads the file to be read into memory starting at the location indicated by the offset. When you include an offset in the R command line, DDT-Z8K displays the address, the file being read, and the next addressable location in memory:

```
-rddt.z8k,00a000000
loading memory at 00A000000 from 'ddt.z8k':
next address= 00A00002D
-
```

8.2.14 The S (Set) Command

The S command allows you to examine and change the contents of bytes, words, or longwords in memory. The forms of the Set command are

```
Ss
SWs
SLs
```

s represents the address in memory to be examined and changed.

DDT-Z8K displays the memory address and its current contents on the following line. In response to the S command, DDT-Z8K displays the address specified in the command line and the contents of memory at that address in byte, word, or longword format. The format used to display the contents of the specified memory address is related to the form of the S command you enter.

You can alter the memory location displayed or leave it unchanged. If you enter a valid hexadecimal value, DDT-Z8K uses that value to replace the contents of the byte, word, or longword at the memory address previously specified. If you do not enter a value, DDT-Z8K leaves the contents of memory unaffected and displays the contents of the next address. In either case, DDT-Z8K continues to display successive memory addresses and values until you enter a period or an invalid value.

8.2.15 The T (Trace) Command

The T command traces program execution for as many as 0FFFFFFFFH program steps. The Trace command has two forms:

```
T
Tn
```

n indicates the number of instructions to execute before returning control to the console.

After DDT-Z8K traces each instruction, it displays the current CPU state and the disassembled instruction in the same form as the X command display.

DDT-Z8K transfers control to the program under test at the address indicated in the PC. If n is not specified, one instruction is executed. Otherwise, DDT-Z8K executes n instructions and displays the CPU state before each step. You can discontinue a long trace before all the steps have been executed by entering any character from the console.

After DDT-Z8K executes a Trace (T) command, it sets the list address used in the L command to the address of the next instruction to be executed.

Note that DDT-Z8K does not trace through a BDOS interrupt instruction, since DDT-Z8K itself makes BDOS calls and the BDOS is not reentrant. Instead, the entire sequence of instructions from the BDOS interrupt through the return from BDOS is treated as one traced instruction.

8.2.16 The U (Untrace) Command

The operation of the U command is identical to the Trace (T) command except that the CPU state is displayed only after the last instruction is executed, rather than after every step. The U command has two forms:

```
U
Un
```

Use n to specify the number of instructions to be executed before DDT-Z8K returns control to the console. You can discontinue the Untrace (U) command before all the steps have been executed by entering any character from the console.

8.2.17 The V (Value) Command

The V command displays information about the last file loaded under DDT-Z8K. The form of the V command is

```
     V
```

The V command's display, shown below, includes the following information for the last file loaded under DDT-Z8K: the file's magic number, the starting address and length of each of the file's segments, the length of the free segment, and the command tail.

```
     magic number=EE03
     code segment start address,length= 0A000000,84B4
     data segment start address,length= 0A0084B4,4122
     bss  segment start address,length= 0A00C5D6,1770
     free segment length= 000020BA
     command tail= ''
```

See Section 3.1 for magic number values and a description of CP/M-8000's command file format.

If you have not already loaded a file by specifying the file's name when you invoked DDT-Z8K or by entering the E or the R command, the V command displays the following error message:

```
     error #90                    file not specified
```

8.2.18 The W (Write) Command

The W command writes the contents of a contiguous block of memory to disk. The Write command has two forms:

```
     Wfilename
     Wfilename,s,f
```

filename is the file specification of the disk file that you want to receive the data. The letters s and f are the first and last addresses of the block to be written. If f does not specify the last address, DDT-Z8K uses the same value that was used for s.

If the first form is used, DDT-Z8K assumes the values for s and f from the last file read with an R command. This form is useful for writing out files after you have installed patches, assuming the overall length of the file is unchanged. If a file was not previously read by an R command, DDT-Z8K displays the following error message in response to the first form of the W command:

```
     error #90                    file not specified
```

If the file specified in the W command already exists on disk, DDT-Z8K deletes the existing file before it writes the new file.

8.2.19 The X (Examine CPU State) Command

The X command displays the entire state of the CPU, including the identifier field of program status area, flag and control word, program counter segment, program counter, all eight data registers, all eight address registers, the disassembled instruction at the memory address currently in the program counter, and the processor flag bits. The X command has two forms:

```
X
Xr
```

Use r to specify one of the general-purpose registers R0 - RF.

The first form of the X command causes DDT-Z8K to display the CPU state as follows:

```
r0=xxxx r1=xxxx r2=xxxx r3=xxxx r4=xxxx r5=xxxx r6=xxxx r7=xxxx
r8=xxxx r9=xxxx ra=xxxx rb=xxxx rc=xxxx rd=xxxx re=xxxx rf=xxxx
id=xxxx fcw=xxxx pcs=xxxx pc=xxxx   L,xxxxxxxx fedcba9876543210
xxxxxxxx: xxxx                    opcode    xxxxx
```

The first two lines DDT-Z8K displays in response to the first form of the X command are the contents of the 16 general purpose registers. The third line displays, from left to right: the identifier field of the program status area (id); the flag and control word (fcw); the program counter segment (pcs); the program counter address (pc); the length, segment and offset of the current instruction; and the individual flags. The fourth line displays the address of the current instruction and the current instruction in hexadecimal, the opcode, and the operand.

The second form, Xr, allows you to display and change the value in the registers of the program being tested. The r denotes the register. DDT-Z8K responds by displaying the current contents of the register, leaving the cursor on that line. If you type a RETURN, the value is not changed. If you type a new valid value and a RETURN, the register is changed to the new value. You can specify the high or low order byte of a register by entering h or l respectively. For example, the command:

```
-Xrh2
```

will cause DDT-Z8K to display the high order byte of register 2. If you type a new value and a RETURN, DDT-Z8K changes the high order byte of register 2 and then displays the low order byte of register 2:

```
-Xrh2
rh2=00 01
rl2=00 02
rh3=00
```

The X command continues to display the high and low order bytes of the registers until you do not type a new value and a RETURN. When you type only a RETURN, the X command displays the entire state of the CPU including any new values you have specified.

8.2.20 The Y (Set/Clear FCW Bits) Command

The Y command allows you to set or clear the status flag and control bits of the FCW. The Y command has two forms:

```
Y
Ybit
```

The first form of the Y command causes DDT-Z8K to display the current hexadecimal value of the Flag and Control Word (FCW) and indicate which bits are set:

```
-Y
1800:  ...vn...........
-
```

The second form of the Y command enables you to specify which FCW bit is to be set or cleared. Figure 8-1 shows the format of the FCW. Note that FCW bits 0, 1, 8, 9, and F are not used with the current implementation of DDT-Z8K. Table 8-3 lists the assignment of each flag status and control bit.

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
0	x	e	v	n	0	0	0	c	z	s	p	d	h	0	0
High-order Byte								Low-Order Byte							

Figure 8-1. Flag and Control Word

Table 8-3. Status Flag and Control Bits.

Word Bit	Status Flag	Assignment
2	h	Half Carry (H)
3	d	Decimal-Adjust (D)
4	p	Parity/Overflow (P/V)
5	s	Sign (S)
6	z	Zero (Z)
7	c	Carry (C)
	Control Bit	
B	n	Non-Vectored Interrupt Enable (NVIE)
C	v	Vectored Interrupt Enabled (VIE)
D	e	Extended Processor Architecture Mode (EPA)
E	x	System/Normal Mode (S/N)

Note that you can also use the X command to set or clear the flag status and control bits when you specify the FCW register. For example, the following X command will clear all flag status and control bits:

```
XFCW
fcw=1800 0000
```

In the example above, DDT-Z8K first displays the current value of the FCW in response to the X command. "fcw=1800" indicates that the v and n control bits are set. All other flag status and control bits are cleared. 0000 is specified for the new value of the FCW; this value causes causes DDT-Z8K to clear all FCW bits.

8.2.21 The $ (Calculate) Command

The $ command allows you to calculate the value of expressions and display the pointers associated with the A, B, D, I, L and S commands. This command also displays the previous value calculated or previous symbol assignment. The forms of the $ command are

```
$
$ <arithmetic expression>
```

The first form of the $ command causes DDT-Z8K to display the pointers associated with the A, B, D, I, L, S, and $$ commands, as well as the value associated with the previous value calculated with the $ command. The command pointers tell you the address where DDT-Z8K would perform the pointers' associated commands were you to invoke a command without specifying an address field. DDT-Z8K displays these pointers and the value last calculated with the $ command as hexadecimal numbers.

The argument to the $ command may be any valid arithmetic expression consisting of numbers of up to eight hexadecimal digits and concatenated with the "+", "-", "*" and "/" operators and parentheses. Use parentheses to specify the evaluation precedence. DDT-Z8K displays the results of the calculation in decimal, hexadecimal, and octal forms.

8.2.22 The $$ (Assign Symbol Value) Command

The $$ command allows you to assign values to symbol names. You can use the symbols you create with the $$ command as operands in many of the other DDT-Z8K command as described below. The form of the $$ command is

```
     $$symbol=value
```

Symbols assigned with the $$ can be up to 11 alphabetical characters in length. Values can be any valid hex address. For example, the following $$ command assigns the symbol name "first" to a value of 0A000000:

```
     $$first=0A000000
     167772160      0A000000      :1200000000
```

In response, the $$ command displays the decimal, hex, and octal forms of the value assigned to the symbol name "first." If you enter a subsequent $ command without specifying a value to be calculated, DDT-Z8K includes the symbols you have assigned with the $$ command in its list of pointers:

```
     -$

     $        00A000000
     $a       000000000
     $b       00A000000
     $d       000000000
     $i       000000000
     $l       00A000000
     $s       000000000
     first    00A000000
```

The symbols you assign with $$ command can be used in place of the operators for the A, B, D, F, G, L, M, S, T, and U command lines. To use a previously defined symbol with any of those commands, specify the symbol by using a $ prefix. Two examples of DDT-Z8k command lines that use symbols assigned through the $$ command are shown below.

```
-D$first

-G$first,$second
```

End of Section 8

Appendix A
Summary of BIOS Functions

Table A-1 lists the BIOS functions supported by CP/M-8000. For more details on these functions, refer to the CP/M-8000 System Guide.

Table A-1. Summary of BIOS Functions

Function	F#	Description
Init	0	Called for Cold Boot
Warm Boot	1	Called for Warm Start
Const	2	Check for Console Character Ready
Conin	3	Read Console Character In
Conout	4	Write Console Character Out
List	5	Write Listing Character Out
Auxiliary Output	6	Write Character to Auxiliary Output Device
Auxiliary Input	7	Read from Auxiliary Input Device
Home	8	Move to Track 00
Seldsk	9	Select Disk Drive
Settrk	10	Set Track Number
Setsec	11	Set Sector Number
Setdma	12	Set DMA Offset Address
Read	13	Read Selected Sector
Write	14	Write Selected Sector
Listst	15	Return List Status
Sectran	16	Sector Translate
Get Memory Region Table Address	18	Address of Memory Region Table

Table A-1. (continued)

Function	F#	Description
Get I/O Byte	19	Get I/O Mapping Byte
Set I/O Byte	20	Set I/O Mapping Byte
Flush Buffers	21	Writes Modified Buffers
Set Exception Vector	22	Sets Exception Vector

Note that CP/M-8000 memory management and context switching functions (_map_addr, _mem_cpy and xfer) are available through BIOS SC #1. Refer to Section 4.2 of the CP/M-8000 System Guide for a description of BIOS SC #1.

End of Appendix A

Appendix B
Transient Program Load Example

This appendix contains an example C langauge program that illustrates how a transient program loads another program with the BDOS Program Load Function (59), but without the CCP.

Listing B-1. CP/M-8000 BDOS Program Loader

```
/****************************************************************
*                                                               *
*               CP/M-8000 BDOS Program Loader                   *
*                                                               *
*               Copyright 1984, Digital Research Inc.           *
*                                                               *
*       This function implements the BDOS Program Load          *
*       Function (59).  The single parameter passed is a        *
*       pointer to a space containing a partially filled        *
*       Load Parameter Block (LPB).   pgmld must fill in the    *
*       base page address and the starting user stack pointer.  *
*       In addition, the Z8000 implementation will set a        *
*       loader flag if the program being loaded requires        *
*       separate I/D space or segmentation.                     *
*                                                               *
*       NOTE:  Unlike the usual CP/M loader, the Z8000 loader   *
*       returns the actual starting address of the code segment *
*       (starting PC) in the LPB, clobbering the program load   *
*       address.  This is because the segment containing the    *
*       code may not be known until load time.                  *
*                                                               *
****************************************************************/

#include "cpmstdio.h"           /* Standard declarations for BDOS, BIOS */

#include "bdosdef.h"            /* Type and structure declarations for BDOS */

#include "biosdef.h"            /* Declarations of BIOS functions       */

#include "basepage.h"           /* Base page structure                  */

#include "xout.h"               /* structure of x.out (".Z8[KS] file")  */

#define SPLIT   0x4000          /* Separate I/D flag for LPB            */
#define SEG     0x2000          /* Segmented code flag for TPA          */
#define NSEG    16              /* Maximum number of x.out segments     */
#define SEGLEN  0x10000         /* Length of a Z8000 segment            */
                                /* Address of basepage (near top of TPA)*/
#define BPLEN   (sizeof (struct b_page))
#define DEFSTACK 0x100          /* Default stack length                 */
#define NREGIONS 2              /* Number of regions in the MRT         */

/* return values */

#define GOOD    0               /* good return value                    */
#define BADHDR  1               /* bad header                           */
#define NOMEM   2               /* not enough memory                    */
#define READERR 3               /* read error                           */
```

Listing B-1. (continued)

```
#define MYDATA  0               /* Argument for map_adr                 *
#define TPAPROG 5               /* Argument for map_adr                 *
#define TPADATA 4               /* Argument for map_adr                 *
                                /* Get actual code segment (as opposed  *
                                /*  to segment where it can be accessed *
                                /*  as data)                            *
#define TRUE_TPAPROG    (TPAPROG | 0x100)

struct lpb                      /* Load Parameter Block                 */
        {
                XADDR   fcbaddr;/* Address of fcb of opened file        */
                XADDR   pgldaddr;/* Low address of prog load area       */
                XADDR   pgtop;  /* High address of prog load area, +1   */
                XADDR   bpaddr; /* Address of basepage; return value    */
                XADDR   stackptr;/* Stack ptr of user; return value     */
                short   flags;  /* Loader control flags; return value   */
        } mylpb;

struct ustack                   /* User's initial stack - nonsegmented  */
        {
                short   two;    /* "Return address" (actually address   */
                                /*  of warm boot call in user's startup)*/
                short   bpoffset;/* Pointer to basepage                 */
        };

struct sstack                   /* User's initial stack - segmented     */
        {
                XADDR   stwo;   /* "Return address" (actually address   */
                                /*  of warm boot call in user's startup)*/
                XADDR   sbpadr; /* Pointer to basepage                  */
        };


struct m_rt {                   /* The Memory Region Table              */
        Int count;
        struct {
                XADDR   tpalow;
                XADDR   tpalen;
        } m_reg[NREGIONS];
};

#define SPREG   1               /* The MRT region for split I/D programs */
#define NSPREG  0               /* The MRT region for non-split programs */
#define SDREG   2               /* The MRT region for split I/D data     */
#define NSDREG  0               /* The MRT region for non-split data     */

#define READ    20              /* Read Sequential BDOS call            */
#define SETDMA  26              /* Set DMA Address BDOS call            */
extern UWORD    bdos();         /* To do I/O into myself (note this     */
                                /*   function does not map 2nd param -  */
                                /*   see mbdos macro below)             */

static XADDR    textloc,        /* Physical locations of pgm sections.  */
                dataloc,
                bssloc,
                stkloc;

static XADDR    textsiz,        /* Sizes of the various sections.       */
                datasiz,
                bsssiz,
                stksiz;
```

Listing B-1. (continued)

```
static UWORD    split,          /* Tells if split I/D or not              */
                seg;            /* Tells if segmented or not              */

static char     *gp;            /* Buffer pointer for char input          */
static char     *mydma;         /* Local address of read buffer           */

struct x_hdr    x_hdr;          /* Object File Header structure           */
struct x_sg     x_sg[NSEG];     /* Segment Header structure               */

static XADDR    segsiz[NSEG];   /* Segment lengths                        */
static XADDR    seglim[NSEG];   /* Segment length limits                  */
static XADDR    segloc[NSEG];   /* Segment base physical addresses        */

static short    textseg,        /* Logical seg # of various segments      */
                dataseg,
                bssseg,
                stkseg;

                /**********************************/
                /*                                */
                /* Start of pgmld function        */
                /*                                */
                /**********************************/

UWORD pgmld(xlpbp)                      /* Load a program from LPB info */
XADDR xlpbp;
{       register int    i,j;            /* Temporary counters etc.      */
        struct m_rt     *mrp;           /* Pointer to a MRT structure   */
        char            mybuf[SECLEN];  /* Local buffer for file reading*/

                                        /* get local LPB copy           */
        cpy_in(xlpbp, &mylpb, (long) sizeof mylpb);

        mydma = mybuf;                  /* Initialize addr for local DMA*/
        gp = &mybuf[SECLEN];            /* Point beyond end of buffer   */

        mrp = (struct m_rt *) bgetseg();/* Get address of memory region */
                                        /*   table (note segment # lost)*/
        if (readhdr() == EOF)           /* Get x.out file header        */
                return (READERR);       /* Read error on header         */

        switch (x_hdr.x_magic)          /* Is this acceptable x.out file*/
        {
          case X_NXN_MAGIC:             /* Non-seg, combined I & D      */
                split = FALSE;
                seg = FALSE;
                break;

          case X_NXI_MAGIC:             /* Non-seg, separate I & D      */
                split = SPLIT;
                seg = FALSE;
                break;

          case X_SX_MAGIC:              /* Segmented - must be combined */
                split = FALSE;
                seg = SEG;
                break;

          default:
                return (BADHDR);        /* Sorry, can't load it!
*/
        }
```

Listing B-1. (continued)

```
/* Set the user space segment number from the low address in the       */
/*  appropriate entry of the MRT.                                       */
/* m_reg[SPREG] is the region used for split I/D programs in the MRT   */
/* m_reg[NSPREG] is used for non-split.                                 */
/* -1            is used for segmented                                  */

/* NOTE -- the tpa limits passed in the LPB are ignored.  This is      */
/*         incorrect, but saves the caller from having to look at the  */
/*         load module to determine the magic number.                   */

        map_adr(seg ? -1L
                    : (mrp->m_reg[split ? SPREG : NSPREG].tpalow),
                0xffff);

        for (i = 0; i < x_hdr.x_nseg; i++) {    /* For each segment...  */
                if( readxsg(i) == EOF)          /* ...get segment hdr   */
                        return(READERR);
                seglim[i] = SEGLEN;             /* ...set max length    */
                segsiz[i] = 0L;                 /* ...and current size  */
        }

                                        /* Set section base addresses   */

        textloc = dataloc = bssloc = stkloc = 0L;

                                        /* Zero section sizes           */
        textsiz = datasiz = bsssiz = 0L;
        stksiz  = DEFSTACK;


        if (seg) {                      /* Locate text & data segments  */
                                        /* if segmented we know nothing */
                textseg = dataseg = bssseg = stkseg = 0;
        } else {                        /* if nonsegmented ...          */
                                        /* assign segment numbers       */
                textseg = 0;
                dataseg = (split) ? 1 : 0;
                stkseg = bssseg = dataseg;

                                        /* assign locations             */
                segloc[textseg] = map_adr(0L, TPAPROG);
                if (split)
                        segloc[dataseg] = map_adr(0L, TPADATA);

                                        /* Assign limits                */
                seglim[textseg] = SEGLEN;
                seglim[dataseg] = mrp->m_reg[split ? SDREG : NSDREG].tpalen
                                  - BPLEN - stksiz;

                                        /* Assign stack location        */
                stkloc = segloc[dataseg] + seglim[dataseg] + stksiz;
        }

        for (i = 0; i < x_hdr.x_nseg; i++)      /* For each segment...  */
                if( (j = loadseg(i)) != GOOD)   /* ...load memory. If   */
                        return (j);             /* error return, pass   */
                                                /* it back.             */
        setbase(setaddr(&mylpb));               /* Set addresses in LPB,*/
                                                /* Set up base page     */
        cpy_out((XADDR) &mylpb, xlpbp, sizeof mylpb);
        return (GOOD);
}
```

Listing B-1. (continued)

```
/* Macro to call BDOS.  First parameter is passed unchanged, second      */
/* is cast into an XADR, then mapped to caller data space.               */

#define mbdos(func, param) (bdos((func), map_adr((XADDR) (param), MYDATA)))


/* Macro to read the next character from the input file (much faster     */
/*   than having to make a function call for each byte)                  */

#define fgetch() ((gp<mydma+SECLEN) ? (int)*gp++&0xff : fillbuff())


/* Routine to fill input buffer when fgetch macro detects it is empty    */


int fillbuf()                           /* Returns first char in buffer */
{                                       /*    or EOF if read fails      */
                                        /* Set up address to read into  */
        mbdos(SETDMA, mydma);
        if (bdos(READ, mylpb.fcbaddr) != 0)      /* Have BDOS do the read*/
                return (EOF);
        gp = mydma;                     /* Initialize buffer pointer    */
        return ((int)*gp++ & 0xff);     /* Return first character       */
}

/* Routine to read the file header */

int readhdr()
{
        register int n, k;
        register char *p;
        p = (char *) &x_hdr;
        for (n = 0; n < sizeof (struct x_hdr); n++) {
                if( (k = fgetch()) == EOF)
                        return (k);
                *p++ = (char) k;
        }
        return (GOOD);
}

/* Routine to read the header for  segment i */

int readxsg(i)
int i;
{
        register int n, k;
        register char *p;

        p = (char *) &x_sg[i];
        for(n = 0; n < sizeof (struct x_sg); n++) {
                if ( (k = fgetch()) == EOF)
                        return (READERR);
                *p++ = (char) k;
        }
        return (GOOD);
}
```

Listing B-1. (continued)

```
/* Routine to load segment number i                                      */
/* This assumes that the segments occur in load order in the file.  It  */
/* assumes that all initialized data and, in the case of combined I/D   */
/* programs, text segments, precede all bss segments.                   */
/*                                                                       */
/* In the case of segmented programs, the stack segment must exist,     */
/* and all segments are presumed to be of maximum length.               */
/* Text, data, bss, and stack lengths are sum of lengths of all such    */
/* segments, and so may be bigger than maximum segment length.          */

int loadseg(i)
int i;
{
        register UWORD l, length;       /* Total, incremental length    */
        register int    type;           /* Type of segment loaded       */
        register short  lseg;           /* logical segment index        */
        register XADDR  phystarg;       /* physical target load address */

        l = x_sg[i].x_sg_len;           /* number of bytes to load      */
        type = x_sg[i].x_sg_typ;        /* Type of segment              */

        lseg  = textseg;                /*   try putting in text space  */

        if (split) {                    /* If separate I/D, this may    */
                switch (type)           /*   be a bad guess             */
                {

          case X_SG_CON:        /* Separate I/D: all data goes  */
          case X_SG_DAT:        /*   in data space              */
          case X_SG_BSS:
          case X_SG_STK:
                lseg  = dataseg;
        }
}

if (seg) {                      /* If segmented, compute phys.  */
                                /* address of segment           */

        /* search to see if seg was used already        */
        /* if so, use the same logical segment index.   */
        /* (if not, loop ends with lseg == i)           */

        for (lseg = 0;
             x_sg[lseg].x_sg_no != x_sg[i].x_sg_no;
             lseg++) ;

        segloc[lseg] = ((long)x_sg[i].x_sg_no) << 24;
}

phystarg = segloc[lseg] + segsiz[lseg]; /* physical target addr */

switch (type)                   /* Now load data, if necessary  */
                                /* save physical address & size */
{
  case X_SG_BSS:                /* BSS gets cleared by runtime  */
                                /*   startup.                   */
        stkloc = (phystarg & 0xffff0000L) + SEGLEN - BPLEN - stksiz;
                                /*  ...in case no stack segment */
        if (bssloc == 0L) bssloc = phystarg;
        bsssiz += l;
        if ((segsiz[lseg] += l) >= seglim[lseg])
                return (NOMEM);
        return (GOOD);                  /* Transfer no data     */
```

Listing B-1. (continued)

```
  case X_SG_STK:                        /* Stack segment:           */
        if (stkloc == 0L) {             /* if segmented, we now     */
                                        /* know where to put        */
                seglim[lseg] -= BPLEN;  /* the base page            */
                stkloc = segloc[lseg] + seglim[lseg];
        }

        stkseg = lseg;
        stksiz += 1;                    /*   adjust size and        */
        seglim[lseg] -= 1;              /*   memory limit           */
        if (segsiz[lseg] >= seglim[lseg])
                return (NOMEM);
        return (GOOD);                  /* Transfer no data         */

  case X_SG_COD:                /* Pure text segment                */
  case X_SG_MXU:                /* Dirty code/data   (better not)*/
  case X_SG_MXP:                /* Clean code/data   (be sep I/D)*/
        if (textloc == 0L) textloc = physt arg;
        textsiz += 1;
        break;

  case X_SG_CON:                /* Constant (clean) data            */
  case X_SG_DAT:                /* Dirty data                       */
        stkloc = (physt arg & 0xffff0000L) + SEGLEN - BPLEN - stksiz;
                                /*  ...in case no stack or          */
                                /*    bss segments                  */
        if (dataloc == 0L) dataloc = segloc[i];
        datasiz += 1;
        break;
}
                                        /* Check seg overflow       */
if ((segsiz[lseg] += 1) >= seglim[lseg])
        return (NOMEM);
                                        /* load data from file      */

/* Following loop is optimized for load speed.          */
/* It knows about three conditions for data transfer:   */
/*  1) Data in read buffer:                             */
/*     Transfer data from read buffer to target         */
/*  2) Read buffer empty and more than 1 sector of      */
/*     data remaining to load:                          */
/*     Read data direct to target                       */
/*  3) Read buffer empty and less than 1 sector of      */
/*     data remaining to load:                          */
/*     Fill read buffer, then proceed as in 1, above    */
```

Listing B-1. (continued)

```
        while (1)                       /* Until all loaded
        {                               /* Data in disk buffer? */
                if (gp < mydma + SECLEN)
                {
                        length = min(l, mydma + SECLEN - gp);
                        cpy_out(gp, physt arg, length);
                        gp += length;
                }
                else if (l < SECLEN)    /* Less than l sector   */
                {                       /*   remains to transfer*/
                        length = 0;
                        mbdos(SETDMA, mydma);
                        fillbuf();
                        gp = mydma;
                }
                else                    /* Read full sector     */
                {                       /*   into target space  */
                        length = SECLEN;
                        bdos(SETDMA, physt arg);
                        bdos(READ, mylpb.fcbaddr);
                }

                physt arg += length;
                l -= length;
        }

        return (GOOD);
}

/* Routine to set the addresses in the Load Parameter Block
/* Unlike normal CP/M, the original load address is replaced on return
/* by the actual starting address of the program (true Code-space addr

int setaddr(lpbp)
struct lpb *lpbp;
{
        register int space;

        space = (split) ? TPADATA : TPAPROG;
        lpbp->pgldaddr = (seg) ? textloc : map_adr(textloc, TRUE_TPAPRO
        lpbp->bpaddr = stkloc;
        lpbp->stackptr = stkloc - (seg? sizeof (struct sstack)

                                            : sizeof (struct ustack));
        lpbp->flags = split | seg;
        return (space);
}

/* Routine to set up the base page.  The parameter indicates whether
 * the data and bss should be mapped in code space or in data space.
 */
```

Listing B-1. (continued)

```
VOID setbase(space)
int space;
{
        struct b_page   bp;

        if (seg) {
                bp.lcode = textloc;
                bp.ltpa  = 0L;
        } else {
                bp.lcode = bp.ltpa = map_adr(textloc, TRUE_TPAPROG);
        }

        bp.htpa = mylpb.stackptr;       /* htpa is where the stack is   */
        bp.codelen = textsiz;

/* was  bp.ldata = dataloc;      rfw */
/*C0*/  bp.ldata = (seg || split) ? dataloc : textloc + textsiz;  /*rfw 5/7/84*/
        bp.datalen = datasiz;

        if (bssloc == 0L) bssloc = dataloc + datasiz;
        bp.lbss = bssloc;
        bp.bsslen = bsssiz;

        bp.freelen = seglim[bssseg] - segsiz[bssseg];

        cpy_out(&bp, map_adr((long) stkloc, space), sizeof bp);
}
```

End of Appendix B

Appendix C
Base Page Format

Table C-1 shows the format of the base page. The base page describes a program's environment. The Program Load Function (59) allocates space for a base page when this function is invoked to load an executable command file. For more details on the Program Load Function and command files, refer to Sections 4.5.7 and 3.0, respectively.

Table C-1. Base Page Format: Offsets and Contents

Offset	Contents
0000 - 0003	Lowest address of TPA (from LPB)
0004 - 0007	1 + Highest address of TPA (from LPB)
0008 - 000B	Starting address of the Text Segment
000C - 000F	Length of Text Segment (bytes)
0010 - 0013	Starting address of the Data Segment (initialized data)
0014 - 0017	Length of Data Segment
0018 - 001B	Starting address of the bss (uninitialized data)
001C - 001F	Length of bss
0020 - 0023	Length of free memory after bss
0024 - 0024	Drive from which the program was loaded
0025 - 0037	Reserved, unused
0038 - 005B	2nd parsed FCB from Command Line
005C - 007F	1st parsed FCB from Command Line
0080 - 00FF	Command Tail and Default DMA Buffer

End of Appendix C

Appendix D
Instruction Set Summary

This appendix contains a summary of the Z8001/2 instruction set used by the ASZ8K assembler. For details on specific instructions, refer to the Z8000 CPU User's Reference Manual, Prentice-Hall, 1982.

Table D-1. Instruction Set Summary

Instruction	Description
adc	Add word with carry
adcb	Add byte with carry
add	Add word
addb	Add byte
addl	Add long word (32 bits)
and	Logical AND word
andb	Logical AND byte
bit	Bit test word
bitb	Bit test byte
call	Call subroutine
calr	Call subroutine, relative
clr	Clear word
clrb	Clear byte
com	1's complement word
comb	1's complement byte
comflg	1's complement flag bits
cp	Logical compare word
cpb	Logical compare byte
cpd	Compare and decrement word
cpdb	Compare and decrement byte
cpdr	Compare, decrement, and repeat word
cpdrb	Compare, decrement, and repeat byte
cpi	Compare and increment word
cpib	Compare and increment byte
cpir	Compare, increment and, repeat word
cpirb	Compare, increment and, repeat byte
cpl	Logical compare longword (32 bits)
cpsd	Compare string and decrement word
cpsdb	Compare string and decrement byte
cpsdr	Compare string, decrement and repeat word
cpsdrb	Compare string, decrement and repeat byte
cpsi	Compare string and increment word
cpsib	Compare string and increment byte
cpsir	Compare string, increment and repeat word
cpsirb	Compare string, increment and repeat byte

Table D-1. (continued)

Instruction	Description
dab	Decimal adjust byte
dbjnz	Decrement byte and jump if not zero
dec	Decrement word
decb	Decrement byte
di	Disable interrupt
div	Divide word
divl	Divide longword (32 bit)
djnz	Decrement word and jump if zero
ei	Enable interrupt
ex	Exchange register contents word
exb	Exchange register contents byte
exts	Extend sign word
extsb	Extend sign byte
extsl	Extend sign longword (32 bit)
halt	Halt processor operation
in	input word
inb	input byte
inc	Increment word
incb	Increment byte
ind	Input word and decrement
indb	Input byte and decrement
indr	Input word, decrement and repeat
indrb	Input byte, decrement and repeat
ini	Input word and increment
inib	Input byte and increment
inir	Input word, increment and repeat
inirb	Input byte, increment and repeat
iret	Interrupt return
jp	Jump
jr	Jump relative
ld	Load word
lda	Load address word
ldar	Load address relative
ldb	Load byte
ldctl	Load Control register
ldctlb	Load Control byte
ldd	Load word and decrement
lddb	Load byte and decrement
lddr	Load word, decrement and repeat
lddrb	Load byte, decrement and repeat
ldi	Load word and increment
ldib	Load byte and increment
ldir	Load word, increment and repeat
ldirb	Load byte, increment and repeat
ldk	Load constant

Table D-1. (continued)

Instruction	Description
ldl	Load longword
ldm	Load multiple words
ldps	Load Program Status
ldr	Load word relative
ldrb	Load byte relative
ldrl	Load longword relative
mbit	Multi-micro bit test
mreq	Multi-micro request
mres	Multi-micro reset
mset	Multi-micro set
mult	Multiply word
multl	Multiply longword
neg	Negate word (2's complement)
negb	Negate byte (2's complement)
nop	No Operation
or	Logical OR word
orb	Logical OR byte
otdr	Output word, decrement and repeat
otdrb	Output byte, decrement and repeat
otir	Output word, increment and repeat
otirb	Output byte, increment and repeat
out	Output word
outb	Output byte
outd	Output word and decrement
outdb	Output byte and decrement
outi	Output word and increment
outib	Output byte and increment
pop	Increment the stack pointer
push	Decrement the stack pointer
res	Reset word
resb	Reset byte
resflg	Reset Flag bit(s)
ret	Return from subroutine
rl	Rotate word left
rlb	Rotate byte left
rlc	Rotate word left through carry
rlcb	Rotate byte left through carry
rldb	Rotate left digit
rr	Rotate word right
rrb	Rotate byte right
rrc	Rotate word right through carry
rrcb	Rotate byte right through carry
rrdb	Rotate right digit

Table D-1. (continued)

Instruction	Description
sbc	Subtract word with carry (borrow)
sbcb	Subtract byte with carry (borrow)
sc	System Call
sda	Shift dynamic arithmetic word
sdab	Shift dynamic arithmetic byte
sdal	Shift dynamic arithmetic longword
sdl	Shift dynamic logical word
sdlb	Shift dynamic logical byte
sdll	Shift dynamic logical longword
set	Set word
setb	Set byte
setflg	Set Flag bit(s)
sin	Special Input byte
sinb	Special Input byte
sind	Special Input word and decrement
sindb	Special Input byte and decrement
sindr	Special Input word, decrement and repe
sindrb	Special Input byte, decrement and repe
sini	Special Input word and increment
sinib	Special Input byte and increment
sinir	Special Input word, increment and repe
sinirb	Special Input byte, increment and repe
sla	Shift word left arithmetic
slab	Shift byte left arithmetic
slal	Shift longword left arithmetic
sll	Shift word left logical
sllb	Shift byte left logical
slll	Shift longword left logical
sotdr	Special word output, decrement and rep
sotdrb	Special byte output, decrement and rep
sotir	Special word output, increment and rep
sotirb	Special byte output, increment and rep
sout	Special word output
soutb	Special byte output
souti	Special word output and increment
soutib	Special byte output and increment
sra	Shift word right arithmetic
srab	Shift byte right arithmetic
sral	Shift longword right arithmetic
srl	Shift word right logical
srlb	Shift byte right logical
srll	Shift longword right logical
sub	Subtract word
subb	Subtract byte
subl	Subtract longword

Table D-1. (continued)

Instruction	Description
tcc	Test Condition Code word
tccb	Test Condition Code byte
test	Test word
testb	Test byte
testl	Test longword
trdb	Translate and decrement
trdrb	Translate, decrement and repeat
trib	Translate and increment
trirb	Translate, increment and repeat
trtdb	Translate, test and decrement
trtdrb	Translate, test, decrement, repeat
trtib	Translate, test and increment
trtirb	Translate, test, increment, repeat
tset	Test and set word
tsetb	Test and set byte
xor	Exclusive OR word
xorb	Exclusive OR byte

End of Appendix D

Appendix E
Error Messages

This appendix lists the error messages returned by the internal components of CP/M-8000 and by the CP/M-8000 programmer's utilities. The sections are arranged alphabetically by the name of the internal component or utility. Error messages are listed alphabetically within each section, with explanations and suggested user responses.

You should contact the dealer from whom you purchased your system if an error persists even after you have followed the suggested responses. In such cases, provide your dealer with the following information:

- Indicate which version of the operating system you are using.
- Describe your system's hardware configuration.
- Provide sufficient information to reproduce the error. Indicate which program was running at the time the error occurred. If possible, you should also provide a disk with a copy of the program.

E.1 AR8K Error Messages

The errors messages returned by AR8K are listed below in alphabetical order with explanations and suggested user responses.

Table E-1. AR8K Error Messages

Message	Meaning
bad file name	Your command line contains an illegal file specification. Refer to Section 1.5 for information on CP/M-8000 file specifications.

Table E-1. (continued)

Message	Meaning
bad read	Your command line includes a file that cannot be read. This message means one of three things: the file to be read is corrupted; a hardware error has occurred; or the file was not correctly written by AR8K when it was created because of an error in the utility's internal logic. Cold start the system and retry the operation. If you receive this error message again, you must erase and recreate the file. Use your backup file, if you maintained one. If the error recurs, check for a hardware error. If the error persists, contact the dealer from whom you purchased your system for assistance. Provide your dealer with the information listed in the beginning of this appendix.
bad write	The disk to which AR8K is writing a file is full. Erase any unnecessary files, or insert a new disk before you reenter the command line.
cannot create filename	The drive code for the file indicated by the variable filename is invalid, or the disk to which AR8K is writing is full. Check the drive code. If the code is valid, the disk is full. Erase any unnecessary files, or insert a new disk before you reenter the command line.
cannot open filename	The file indicated by the variable filename cannot be opened because the filename or the drive code is incorrect. Check the drive code and the filename before you reenter the command line.

Table E-1. (continued)

Message	Meaning
header write error	The operating system returned an error while AR8K was writing an archive header to the archive. Check that the disk is not full. If the disk is full, erase any unnecessary files from it, or replace it with another disk that contains a copy of the archive.
No key specified	Your command line did not contain an AR8K key. Reenter the command line and specify which action is to be performed by AR8K. AR8K will display its command line keys when you type AR8K and enter a RETURN. See Section 7 for a detailed explanation of the AR8K command line.
not archive format: filename	The file indicated by the variable filename is not a library. Ensure that you are using the correct filename before you reenter the command line.
not object file: filename	The file indicated by the variable filename is not an object file and cannot be added to the library. Any file added to the library must be an object file, output by the assembler, ASZ8K, or the compiler. Assemble or compile the file before you reenter the AR8K command line.
only one of keys [drqtx] allowed	The AR8K command line requires one of the D, R, Q, T, or X command keys, but not more than one. Reenter the command line with the correct command. Refer to Section 7 for an explanation of the AR8K commands.

Table E-1. (continued)

Message	Meaning
filename not in library	The object module indicated by the variable filename is not in the library. Ensure that you are requesting the filename of an existing object module before you reenter the command line.
temp file write error	The disk to which AR8K was writing the temporary file is full. Erase any unnecessary files, or insert a new disk before you reenter the command line.
Too many file names	You command line specifies more than 256 files. The upper limit for file names in an AR8K command line is 256. Delete one or more file names from the command line and reenter it.
Unable to recopy the archive	AR8K completed the working copy of the archive but the operating system returned an error when AR8K attempted to overwrite the original archive with the working copy. Ensure that the original archive file is set to DIR, R/W status and that the disk to be written to is not full.
unknown key	The key in the command line is an invalid option. Enter AR8K and RETURN and the utility displays a usage statement indicating valid command line keys. Refer to Section 7 for an explanation of the command line options. Specify a valid option and reenter the command line.

Table E-1. (continued)

Message	Meaning
usage: AR8K KEY ARCFILE [files. . .]	This message indicates a syntax error in the command line. The correct format for the command line is given, with KEY indicating one of the command letters D, R, Q, T, or X and the file specification for the object file(s) in brackets. Refer to Section 7 for a more detailed explanation of the command line.

E.2 ASZ8K Error Messages

The CP/M-8000 assembler, ASZ8K, returns both diagnostic error codes and fatal error messages. Fatal errors stop the assembly of your program.

E.2.1 ASZ8K Diagnostic Error Codes

Diagnostic error codes report errors in the syntax and context of the program being assembled, without interrupting assembly. Refer to the Zilog 16-Bit Microprocessor User's Manual for a full discussion of the assembly language syntax.

When ASZ8K encounters errors within the assembly language program, it lists them as single-character codes in the leftmost position of the source listing. ASZ8K also echoes the line in error to the console so that you need not examine the source listing to determine if errors are present. Table E-2 defines the ASZ8K error codes.

Table E-2. ASZ8K Diagnostic Error Codes

Code	Meaning
A	Macro definition syntax error
C	Incorrect placement of an .ELSE or .ENDIF directive

Table E-2. (continued)

Code	Meaning
E	Expression evaluation error: too large a val for a field; Register 0 specified fc addressing mode where restricted; register pai used as a pointer in nonsegmented mode register used as a pointer in segmented mode; relocatable value used in an expression tha allows only constants
L	Label error: label field missing from an .EQU .COMMON, .SECT, or .MACRO directive
M	Multiple definition of a symbol: a symbol i the indicated source line has been defined a both global and common; delete one of th directives and reassemble the source file
O	Opcode error: undefined opcode or pseudo-op improperly placed .EXIT directive; imprope operand format for specified opcode
P	Phase error: the type, value, or relocatio type of a label were inconsistent between th two passes of the assembler; this error migh be caused by label redefinition or tw indistinguishable labels
S	Syntax error: invalid characters or misplace delimiters in pseudo-op, expression, or operan field following last assembled operand
U	Undefined symbol: label operand in this statement has not appeared elsewhere in a statement that generates machine code or reserves memory, as in an .EQU or .ORG or .SET directive

E.2.2 ASZ8K Fatal Error Messages

The fatal error messages displayed by ASZ8K are listed in Table E-3. When an error occurs because the disk is full, ASZ8K creates a partial file. You should erase the partial file to ensure that you do not try to link it.

Table E-3. ASZ8K Fatal Error Messages

Message	Meaning
Cannot create filename	The operating system returned an error to ASZ8K while the utility was attempting to write the file indicated by the variable filename on the disk. Check the filename, drive code, user number, and level of disk write protection. You may also have to erase a file to make space in the directory. Respecify the command line before you reassemble the source file.
Cannot open filename	The file indicated by the variable filename does not exist, is invalid, or has an invalid drive code or user number. Check the filename, drive code, and user number. Respecify the command line before you reassemble the source file.
Cannot reopen vm file	The operating system returned an error to ASZ8K when the assembler attempted to reopen its work file. Erase the work file and check the number of directory entries and space remaining on the disk. Reassemble the source file.
Input stack overflow	The source code contains too many operations for the user stack -- the program is too large to be assembled.
No PREDEF file	ASZ8K cannot locate the ASZ8K.PD predefinition file. Check that this file is present on the default disk and in the correct user area. If ASZ8K.PD is not on the disk, recopy it from your distribution disk.

Table E-3. ASZ8K Fatal Error Messages

<table>
<tr><th>Message</th><th>Meaning</th></tr>
<tr><td>PREDEF file error at line number</td><td>The predefinition file ASZ8K.PD has been corrupted. This message indicates the line number with ASZ8K.PD that contains the bad data. Erase the corrupted copy of ASZ8K.PD from the disk and replace it with a new copy from your distribution disk.</td></tr>
<tr><td>Too many externals</td><td>The source code uses too many externally defined global symbols for the size of the external symbol table. Eliminate some externally defined global symbols and reassemble the source file.</td></tr>
<tr><td>Usage: asz8k [-o outfile] [-luxs] file.8k{n|s}</td><td>This error message indicates that your ASZ8K command line contains a syntax error. This message shows you the format of the ASZ8K command line as described in Section 5.2.</td></tr>
</table>

E.3 BDOS Error Messages

The CP/M-8000 Basic Disk Operating System, BDOS, returns fatal error messages at the console. The BDOS error messages are listed below in alphabetic order with explanations and suggested user responses.

Table E-4. BDOS Error Messages

Message	Meaning
CP/M Disk change error on drive x	The disk in the drive indicated by x is not the same disk the system logged in previously. When the disk was replaced you did not enter a Ctrl-C to log in the current disk. Therefore, when you attempted to write to, erase, or rename a file on the current disk, the BDOS set the drive status to read-only and warm booted the system. The current disk in the drive was not overwritten. The drive status was returned to read-write when the system was warm booted. Each time a disk is changed, you must type a Ctrl-C to log in the new disk.
CP/M Disk file error: filename is Read-Only. Do you want to: Change it to read/write (C), or Abort (A)?	You attempted to write to, erase, or rename a file whose status is read-only. Specify one of the options enclosed in parentheses. If you specify the C option, the BDOS changes the status of the file to read/write and continues the operation. The read-only protection previously assigned to the file is lost. If you specify the A option or a CTRL-C, the program terminates and CP/M-8000 returns the system prompt.
CP/M Disk read error on drive x WARNING -- Do not attempt to change disks Do you want to: Abort (A), Retry (R), or Continue with bad data (C)?	This message indicates a hardware error. Specify one of the options enclosed in parentheses. If you specify the A option or enter Ctrl-C, CP/M-8000 terminates the operation and returns the system prompt.

Table E-4. (continued)

Message	Meaning
	If you specify the R option, CP/M-8000 retries the operation. If this action fails, the system reprompts with the option message. Specifying option C causes the system to ignore the error and continue program execution. Use this option with caution. Program execution should not be continued for some types of programs. For example, if you are updating a data base and receive this error but continue program execution, you can corrupt the index fields of the entire data base. For other programs, continuing program execution is recommended. For example, when you transfer a long text file and receive an error because one sector is bad, you can continue transferring the file. Once the file is transferred, review it and add the data that was not transferred because of the bad sector.
CP/M Disk write error on drive x Do you want to: Abort (A), Retry (R), or Continue with bad data (C)?	This message indicates a hardware error. Specify one of the options enclosed in parentheses. Please see the explanation of the preceding error message for a description of each option presented in this message.
CP/M Disk select error on drive x Do you want to: Abort (A), Retry (R)	There is no disk in the drive, or the disk is not inserted correctly. Ensure that the disk is securely inserted in the drive. If you enter the R option, the system retries the operation. If you enter the A option or Ctrl-C the program terminates and CP/M-8000 returns the system prompt.

Table E-4. (continued)

Message	Meaning
CP/M Disk select error on drive x	The disk selected in the command line is outside the range A through P. CP/M-8000 can support up to 16 drives, lettered A through P. Check the documentation provided by the manufacturer to find out which drives your particular system configuration supports. Specify the correct drive code and reenter the command line.

E.4 BIOS Error Messages

The CP/M-8000 BIOS error messages are listed below in alphabetical order with explanations and suggested user responses.

Table E-5. BIOS Error Messages

Message	Meaning
BIOS ERROR -- DISK X NOT SUPPORTED	The disk drive indicated by the variable X is not supported by the BIOS. The BDOS supports a maximum of 16 drives, lettered A through P. Check the manufacturer's documentation for your system configuration to find out which of the BDOS drives your BIOS implements. Specify the correct drive code and reenter the command line.
BIOS ERROR -- Invalid Disk Status	The disk controller returned unexpected or incomprehensible information to the BIOS. Retry the operation. If the error persists, check the hardware.

E.5 CCP Error Messages

The CP/M-8000 Console Command Processor, CCP, returns two types of error messages at the console: diagnostic and internal logic error messages.

E.5.1 Diagnostic Error Messages

The CCP error messages are listed below in alphabetical order with explanations and suggested user responses.

Table E-6. CCP Diagnostic Error Messages

Message	Meaning
bad relocation information bits	This message is a result of a BDOS Program Load Function (59) error. It indicates that the file specified in the command line is not a valid executable command file, or that the file has been corrupted. Ensure that the file is a command file. Section 3 of this manual describes the format of a command file. If the file has been corrupted, reassemble or recompile the source file, and relink the file before you reenter the command line.
File already exists	This error occurs during a REN command. The name specified in the command line as the new filename already exists. Use the ERA command to delete the existing file if you wish to replace it with the new file. If not, select another filename and reenter the REN command line.

Table E-6. (continued)

Message	Meaning
insufficient memory or bad file header	This error could result from one of three causes: ● The file is not a valid executable command file. Ensure that you are requesting the correct file. This error can occur when you enter the filename before you enter the command for a utility. Check the appropriate section of this manual or the CP/M-8000 User's Guide for the correct command syntax before you reenter the command line. If you are trying to run a program when this error occurs, the program file may be corrupted. Reassemble or recompile the source file and relink the file before you reenter the command line. ● The program is too large for the available memory. Add more memory boards to the system configuration, or rewrite the program to use less memory. ● The program is linked to an absolute location in memory that cannot be used. The program must be made relocatable or linked to a usable memory location. The BDOS Get/Set TPA Limits Function (63) returns the high and low boundaries of the memory space that is available for loading programs.

Table E-6. (continued)

Message	Meaning
No file	The filename specified in the command line does not exist. Ensure that you use the correct filename and reenter the command line.
No wildcard filenames	The command specified in the command line does not accept wildcards in file specifications. Retype the command line using a specific filename.
read error on program load	This message indicates a premature end-of-file. The file is smaller than the header information indicates. Either the file header has been corrupted or the file was only partially written. Reassemble or recompile the source file, and relink the file before you reenter the command line.
SUB file not found	The file requested either does not exist or does not have a filetype of SUB. Ensure that you are requesting the correct file. Refer to the section on SUBMIT in the CP/M-8000 User's Guide for information on creating and using submit files.
Syntax: REN newfile=oldfile	The syntax of the REN command line is incorrect. The correct syntax is given in the error message. Enter the REN command followed by a space, then the new filename, followed immediately by an equals sign (=) and the name of the file you want to rename.

Table E-6. (continued)

Message	Meaning
Too many arguments: argument?	The command line contains too many arguments. The extraneous arguments are indicated by the variable argument. Refer to the CP/M-8000 User's Guide for the correct syntax for the command. Specify only as many arguments as the command syntax allows and reenter the command line. Use a second command line for the remaining arguments, if appropriate.
User # range is [0-15]	The user number specified in the command line is not supported by the BIOS. The valid range is enclosed in the square brackets in the error message. Specify a user number between 0 and 15 (decimal) when you reenter the command line.

E.5.2 CCP Internal Logic Error Messages

The following message indicates an undefined failure of the BDOS Program Load Function (59).

Program Load Error

If you receive this message, contact the dealer from whom you purchased your system for assistance. You should provide the following information:

- Indicate which version of the operating system you are using.
- Describe your system's hardware configuration.
- Provide sufficient information to reproduce the error. Indicate which program was running at the time the error occurred. If possible, you should also provide a disk with a copy of the program.

E.6 DDT-Z8K Error Messages

Error messages for the CP/M-8000 debugger, DDT-Z8K, are listed be in alphabetical order.

Table E-7. DDT-Z8K Error Messages

Message	Meaning
arg1 > arg2	The starting address in a D (Display), F (Fill), L (List), or W (Write) command line is greater than the final address. Respecify the command.
arg1 >= arg2	The starting address in a D (Display), F (Fill), L (List), or W (Write) command line is greater than or equal to the final address. Respecify the command line.
bad argx	DDT-Z8K displays this error message when you have specified an invalid argument in a DDT-Z8K command. The actual error message indicates which argument in your command line is invalid by replacing x with the appropriate number.
bad character in symbol	This error message indicates that the symbol name to which you are assigning a value with the $$ command contains a delimiter character. See Section 1.5 for a list of the CP/M-8000 delimiter characters.

Table E-7. (continued)

Message	Meaning
bad delimiter	DDT-Z8K displays this error message when the operands or arguments in your command line are not correctly separated. Most DDT-Z8K commands accept a comma or space as a delimiter character. See the DDT-Z8K command descriptions in Section 8.
bad header	This message occurs in response to an E (Load for Execution) command. The error could be caused by one of the following conditions: ● The system you are using does not have enough memory available. Ensure that the program and DDT-Z8K fit into the TPA. Exit DDT-Z8K. Use the SIZE68 Utility to display the amount of space your program occupies in memory. DDT-Z8K is approximately 20K bytes. The BDOS Get/Set TPA Limits Function (63) returns the high and low boundaries of the TPA. If you do not have sufficient space in the TPA to execute your command file and DDT-Z8K simultaneously, additional memory must be added to the system configuration. ● The file is not a command file or has a corrupted header. If the command file does not run, but you are sure that your memory space is adequate, use the R command to look at the file and check its format. You might be trying to debug a file that is not a command file. If it is a command file, the header may have been corrupted. Reassemble or recompile the source file before you reenter the E command line.

Table E-7. (continued)

Message	Meaning
	● The command file you are debugging is linked to an absolute location in memory that is already occupied by DDT-Z8K. DDT-Z8K is approximately 20K bytes, and usually resides in the highest addresses of the TPA. The recommended location for linking your file is the base address of the TPA + 100H. The BDOS Get/Set TPA Limits Function (63) returns the high and low boundaries of the TPA.
bad opcode	This error occurs in response to a List (L) command if the memory location being disassembled does not contain a valid instruction. The error may have been caused by one of three things: ● You gave the L command the wrong address. Reenter the L command with the correct address. ● The file is not a command file. Ensure that the file you specify is a command file and reenter the L command. ● The command file has been corrupted. Reassemble or recompile the source file before you reread it into memory with a Load for Execution (E) or Read (R) command, as appropriate. If the problem persists, use the debugging commands in DDT-Z8K to look for an error in the program that causes it to overwrite itself. See Section 8 for a complete description of the DDT-Z8K commands and options.

Table E-7. (continued)

Message	Meaning
can't load program	This message occurs in response to a Load for Execution (E) command. Possible causes for this error condition include an incorrect user number, drive code, or filename. Check the user number, drive code, and filename before you reenter the command line. If the error persists, it is probably due to a premature end-of-file mark -- the file is smaller than the header information indicates. Either the file has been corrupted or the file was only partially written. Reassemble or recompile the source file and relink the file before you reissue the command line.
can't open inload file	This error occurs during a Read (R) command. It indicates an incorrect user number, drive code, or filename. Check the user number, drive code, and filename before you reenter the command line.
can't open memory file	This error occurs during a Write (W) command. The disk to which DDT-Z8K is writing has no more directory space available: in effect, the disk is full. If you have another drive available, reenter the Write (W) command and direct the file to the disk on that drive. If you do not have another drive available, you must exit DDT-Z8K and lose the contents of memory. Erase any unnecessary files, or insert a new disk.

Table E-7. (continued)

Message	Meaning
file not specified	DDT-Z8K displays this error message in response to an R (Read) or W (Write) command line that does not include a filename. Specify a filename and then reenter either command. DDT-Z8K also displays this error message in response to a V (Value) command when you have not previously loaded a file. Load a file with the E (Load for Execution) or R (Read) command before you reenter the V command. You can also load a file under DDT-Z8K by specifying the filename when you invoke the debugger.
memory file write error	The disk to which DDT-Z8K is writing is full, or the disk contains a bad sector. Retry the command. If the error persists, and you have another disk drive available, redirect the output to the disk on that drive. If you do not have another drive available, you must exit DDT-Z8K. Use the STAT command to check the space on the disk. If it is full, erase any unnecessary files, or insert a new disk. Because the contents of memory are lost when you exit DDT-Z8K, you must reload the file in memory. If the disk was not full, it has a bad sector, and you should replace it.
memory overflow	This message occurs during a Read (R) command when the file being read is too large to fit in memory. DDT-Z8K reads only the portion of the file that can be read into the existing memory. To debug this program, additional memory boards must be added to the system configuration.

Table E-7. (continued)

Message	Meaning
read error	This message indicates one of three error situations. Retry the operation -- if the error persists, try the responses indicated: • A write error at the time the file was created. You must recreate the file. If the error recurs, or if you cannot write to the disk, the disk is bad. • A bad disk. Use PIP or COPY to copy the file from the bad disk to a new disk. Any files that cannot be copied must be recreated or replaced from backup files. Discard the damaged disk. • A hardware error. If the error persists, check your hardware.
unknown flag mnemonic	The flag specified in an Y command line is invalid. Check that you have typed the correct flag mnemonic and reenter the command. Table 8-3 lists the valid status flag and control bit mnemonics.
unknown instruction	This error occurs during a List (L) command. The instruction being disassembled has an illegal value. Use a Display (D) command to display the location of the error. This error could be caused by one of three things: • The memory location being disassembled does not contain an instruction. Ensure that the area selected is an instruction. If it is not, reenter the L command with a correct location.

Table E-7. (continued)

Message	Meaning
	● The size field of the instruction has been corrupted. Use the debugging commands in DDT-Z8K to look for an error that causes the program to overwrite itself. Refer to Section 8 for a complete description of the DDT-Z8K commands and options. ● An invalid instruction was generated by the compiler or assembler used to create the program. Recompile or reassemble the source file before you reinvoke DDT-Z8K.
unknown register	The register specified in an X command line is invalid. Reenter the command line with a legal register. See Section 8.2.19.

E.7 DUMP Error Messages

DUMP returns fatal and diagnostic error messages at the console. The DUMP error messages are listed below in alphabetical order with explanations and suggested user responses.

Table E-8. DUMP Error Messages

Message	Meaning
Unable to open filename	Either the drive code for the input file indicated by the variable filename is incorrect, or the filename is misspelled. Check the filename and drive code before you reenter the DUMP command line.

Table E-8. (continued)

Message	Meaning
Usage: dump [-shhhhhh] file	The command line syntax is incorrect. The correct syntax is given in the error message. Specify the DUMP command and the filename. If you want to display the contents of the file from a specific address in the file, specify the -S option followed by the address. Refer to Section 7.2 for a description of the DUMP command line and options.

E.8 LD8K Error Messages

The CP/M-8000 Linker, LD8K, returns the error messages listed below.

Table E-9. LD8K Error Messages

Message	Meaning
bad read in filename	The object file indicated by the variable filename does not contain the number of bytes indicated by the file's header. The file is either incorrectly formatted or has been corrupted. This error is commonly caused when the input to LD8K is a partially assembled or compiled object file. ASZ8K and ZCC create partial object files when a "disk full" status is returned to them by the operating system while assembling or compiling a file. Use the DUMP utility to ensure that the file is a complete object file. Reassemble or recompile the source before you relink the file.

Table E-9. (continued)

Message	Meaning
bad relocation read in filename	LD8K could not read the relocation information in the file specified by filename. The file has been corrupted. Reassemble or recompile the files source before you relink it.
bad segment header read in filename	The specified file contains an error in its segment header. The file may have been damaged. Reassemble or recompile the file before you reenter the LD8K command line.
bad symbol table read in filename	LD8K detected an error in the specified file's symbol table or symbol table header entry. The file may have been damaged. Reassemble or recompile the source before you relink the file.
bad write in filename	LD8K could not write the file specified by filename. The disk or disk file directory may be full. Erase any unnecessary files or insert a new disk before reentering the LD8K command line.
code too big for non-seg load	The LD8K command line specified a nonsegmented load but the code to be loaded exceeded 64 Kbytes.

Table E-9. (continued)

Message	Meaning
cannot create filename	LD8K was unable to create, or open to write, the file indicated by filename. Either the command line specified an invalid drive code, or the disk to which LD8K was writing is full. If the drive code was correct, the disk is full. Erase any unnecessary files, or insert a new disk before you reenter the command line.
cannot open filename	LD8K cannot open the specified file to read. Either the specified filename is invalid, or the file does not exist. Check the filename before you reenter the command line.
<filename> is a segmented file in a non-segmented load	The first file specified in the command line was nonsegmented, but a subsequent file was segmented. The two types of object files cannot be intermixed. LD8K determines the type of load from the first file it reads. Determine the type of load you want performed, and reenter a command line that specifies files of the same type.
hash table overflow: number	This error message indicates a fatal error in the internal logic of LD8K. Recopy LD8K from the distribution disk to ensure that you have an undamaged copy.
map overflow in filename	The load caused LD8K to overflow its internal map table. LD8K allows 512 map entries per load. The specified file contains the entry that caused the map entry count to exceed 512.

Table E-9. (continued)

Message	Meaning
multiple def: list of symbols	The list of symbols specifies those global symbols in the load which were defined more than once. Rewrite the source code. Provide a unique definition for each symbol and reassemble or recompile the source code before you relink the file.
not x.out format: filename	The specified input file is either not in the proper object file format for CP/M-8000 or the file has been corrupted. Ensure that the file is an object file, output by ASZ8K or the compiler (ZCC). Section 3 describes the CP/M-8000 x.out file format. You may have to reassemble or recompile the file before you relink it.
segment overflow in filename	The file specified by filename contains code for an undefined segment. Rewrite the source so that the file uses a valid segment. Reassemble or recompile the source before you relink the file.
segment number overflow in filename	The load attempted to place too much code or data into the segment specified by number. 64K is the maximum segment capacity. The error occurred when LD8K attempted to load the file specified by filename.
stack segment too big	The stack segment specification exceeds 64K. Rewrite the source code so that it specifies a stack segment of less than 64K. Reassemble or recompile the source before you relink the file.

Table E-9. (continued)

Message	Meaning
symap overflow in filename	The load caused LD8K to overflow its symbol map. LD8K allows 4096 symbols per load. The specified file contains the symbol that caused the symbol map count to exceed 4096. Rewrite the source code so that the file contains fewer symbols. Reassemble or recompile the source before you relink the file.
symbol seek failed in filename	LD8K detected a symbol error in the file specified by filename. The file may have been damaged. Reassemble or recompile the source before you relink the file.
symbol table overflow in filename	The load caused the LD8K symbol table to overflow. LD8K allows a maximum of 6114 unique global symbols. The file specified by filename contains the symbol that caused the unique global symbol count to exceed 6114. You can resolve this error by rewriting the source code so that the file declares fewer global symbols. Reassemble or recompile the source before you relink the file.
too many files	The command line specified more than 128 filenames. Reenter a command line that specifies fewer than 128 filenames.
too many modules at filename	More than 256 modules appeared in the specified object file during the link. Rewrite the source code so that the file contains fewer modules. Reassemble or recompile the source before you relink the file.

Table E-9. (continued)

Message	Meaning
number undefined symbols: symbol list	This message indicates the number and provides a list of the symbols that were not defined. Provide a valid definition for these symbols and reassemble the source code before you reenter the LD8K command line.
unexpected EOF while reading header on filename	LD8K could not properly read the file specified by filename. The file has been damaged. Reassemble or recompile the file's source before you relink it.
unknown option char	The command line contains an option that is not allowed. The option noted by char will be ignored.

E.9 NMZ8K Error Messages

NMZ8K returns fatal diagnostic error messages at the console. The NMZ8K error messages are listed below in alphabetical order with explanations and suggested user responses.

Table E-10. NMZ8K Error Messages

Message	Meaning
cannot open filename	The file specification indicated by the variable filename is incorrect. Check the spelling of the filename and the drive code before you reenter the command line.

Table E-10. (continued)

Message	Meaning
Not x.out format: filename	The input file indicated by the variable filename is neither an object file nor a command file. This message can also indicate that the file has been corrupted. NMZ8K prints the symbol table of an object file or a command file. Ensure that the file is one of these types of file. If the file is an object or command file and you receive this message, the file is corrupted. Rebuild the file with the compiler or assembler. If the file is a command file, relink it. Reenter the NMZ8K command line.
read error on file: filename	The input file indicated by the variable filename is truncated. Rebuild the file with the compiler or assembler. If the file is a command file, relink it. Reenter the NMZ8K command line.
usage: nmz8k objectfile	The command line syntax is incorrect. Use the syntax given in the error message and reenter the command line.

E.10 SIZEZ8K Error Messages

SIZEZ8K returns fatal, diagnostic error messages at the console. The SIZEZ8K error messages are listed below in alphabetical order with explanations and suggested user responses.

Table E-11. SIZEZ8K Error Messages

Message	Meaning
cannot open filename	Either the drive code is incorrect, or the file indicated by the variable filename does not exist. Check the drive code and filename. Reenter the SIZEZ8K command line.
Not x.out format: filename	The file indicated by the variable filename is neither an object file nor a command file. SIZEZ8K requires either an object file, output by the assembler or the compiler, or a command file, output by the linker. Ensure that the file specified is one of these and reenter the SIZEZ8K command line.
read error on filename	The file indicated by the variable filename is truncated. Rebuild the file. Reassemble or recompile, and relink the source file before you reenter the SIZEZ8K command line.

E.12 XDUMP Error Messages

The XDUMP utility returns fatal error messages at the console. XDUMP error messages are listed below in alphabetical order with explanations and suggested user responses.

Table E-12. XDUMP Error Messages

Message	Meaning
cannot open filename	Either the drive code is incorrect, or the file indicated by the variable filename does not exist. Check the drive code and filename. Reenter the SIZEZ8K command line.
Not x.out format filename	The file indicated by the variable filename is neither an object file nor a command file. SIZEZ8K requires either an object file, output by the assembler or the compiler, or a command file, output by the linker. Ensure that the file specified is one of these and reenter the SIZEZ8K command line.
read error on filename	The file indicated by the variable filename is truncated. Rebuild the file. Reassemble or recompile, and relink the source file before you reenter the SIZEZ8K command line.

End of Appendix E

Appendix F
New Functions and Implementation Changes

CP/M-8000 has six new Basic Disk Operating System (BDOS) functions and additional implementation changes in the BDOS functions and data structures that differ from other CP/M systems.

Table F-1. New BDOS Functions

Function	Number
Get Free Disk Space	46
Chain to Program	47
Flush Buffers	48
Set Exception Vector	61
Set Supervisor State	62
Get/Set TPA Limits	63

F.1 BDOS Function and Data Structure Changes

Implementation changes in CP/M-8000 BDOS functions are described in Table F-2. Data structure changes in CP/M-8000 are described in Table F-3.

Table F-2. BDOS Function Implementation Changes

BDOS Function	Number	Implementation Change
Return Version Number	12	Contains the version number 3022H, indicating CP/M-8000 Version 1.1.
Reset Disk System	14	Does not log in drive A when it resets the disk system.
Open File	15	Opens a file only at extent 0, the base extent.
Get Disk Parameters	31	Returns a copy of the Disk Parameter Block (DPB).

Table F-3. BDOS Data Structure Implementation Changes

Structure	Implementation Change
Base Page	Additional information has been added. The base page is no longer located at a fixed address. Appendix C outlines the structure of the base page.
File Control Block	The byte sequence for the Random Record Field has changed. The most significant byte (r0) is first and the least significant byte (r2) is last.

F.2 BDOS Functions Not Supported By CP/M-8000

The list below contains functions and commands supported by other CP/M systems, but not by CP/M-8000.

Table F-4. BDOS Functions Not Supported by CP/M-8000

BDOS Function	Number
Get Address of Allocation Vector	27
Set DMA Base	51
Get DMA Base	52
Get Maximum Memory*	53
Get Absolute Memory*	54
Allocate Absolute Memory*	55
Free Memory*	56
Free All Memory*	57

* Memory management within CP/M-8000 does not require these functions.

End of Appendix F

Appendix G
Decimal-ASCII-Hex Table

DECIMAL	ASCII	HEX	DECIMAL	ASCII	HEX	DECIMAL	ASCII	HEX
0	NUL	00	43	+	2B	86	V	56
1	SOH	01	44	,	2C	87	W	57
2	STX	02	45	-	2D	88	X	58
3	ETX	03	46	.	2E	89	Y	59
4	EOT	04	47	/	2F	90	Z	5A
5	ENQ	05	48	0	30	91	[	5B
6	ACK	06	49	1	31	92	\	5C
7	BEL	07	50	2	32	93	]	5D
8	BS	08	51	3	33	94	^	5E
9	HT	09	52	4	34	95		5F
10	LF	0A	53	5	35	96	`	60
11	VT	0B	54	6	36	97	a	61
12	FF	0C	55	7	37	98	b	62
13	CR	0D	56	8	38	99	c	63
14	SO	0E	57	9	39	100	d	64
15	SI	0F	58	:	3A	101	e	65
16	DLE	10	59	;	3B	102	f	66
17	DC1	11	60	<	3C	103	g	67
18	DC2	12	61	=	3D	104	h	68
19	DC3	13	62	>	3E	105	i	69
20	DC4	14	63	?	3F	106	j	6A
21	NAK	15	64	@	40	107	k	6B
22	SYN	16	65	A	41	108	l	6C
23	ETB	17	66	B	42	109	m	6D
24	CAN	18	67	C	43	110	n	6E
25	CR	19	68	D	44	111	o	6F
26	SUB	1A	69	E	45	112	p	70
27	ESC	1B	70	F	46	113	q	71
28	FS	1C	71	G	47	114	r	72
29	GS	1D	72	H	48	115	s	73
30	RS	1E	73	I	49	116	t	74
31	US	1F	74	J	4A	117	u	75
32	SP	20	75	K	4B	118	v	76
33	!	21	76	L	4C	119	w	77
34	"	22	77	M	4D	120	x	78
35	#	23	78	N	4E	121	y	79
36	$	24	79	O	4F	122	z	7A
37	%	25	80	P	50	123	{	7B
38	&	26	81	Q	51	124	|	7C
39	'	27	82	R	52	125	}	7D
40	(	28	83	S	53	126	~	7E
41	)	29	84	T	54	127	DEL	7F
42	*	2A	85	U	55			

End of Appendix G

Index